**Chapter 11. Applying the Patterns: A Transformation Design Story, Part 1**

*This is a detailed design for a cloud native transformation. Here we will lay out patterns from start to finish, explaining the order and reason for the choices as we go along. It’s a lengthy and involved process, so we have divided it into two chapters. This one covers the period from pre-initiative prep through the point where we have researched, experimented, and prototyped until successfully uncovering the likely best transformation path. Part 2 picks up with verifying the path and beginning to build a production-ready cloud native platform, carries through onboarding everyone onto the new system with a new way of working, and then moves to shutting down the old one.*

For several chapters now we have watched WealthGrid struggle with multiple attempts at a cloud native transformation, only to fail each time. The story we are going to tell now is how to do it right—in other words, what WealthGrid would have done, had they known better. We will be using patterns to show the way, so that now *you* will know better.

First, let’s take a quick review of the story thus far.

WealthGrid’s first erroneous attempt to move to cloud native is an extremely common strategy: treating the transformation as only a minor technology shift. Many companies try exactly this, and equally many fail at it. The technology is new and rests upon a complex distributed architecture, so right away you have two things nobody at WealthGrid (or pretty much anybody at any other company, aside from a handful of tech giants) really understands or has any experience with. Setting a small team with no background in the tech to deliver a full transformation as a side project while also doing their usual work on the existing system? We saw how well that worked for WealthGrid—or, rather, didn’t work.

WealthGrid’s people, Jenny and Steve, as well as others, saw it too. And they did try to correct course. For their second attempt they tried going all in, assigning a big team (and big budget) to bringing the whole organization onto the cloud. Unfortunately, this all-hands-on-deck approach did not work any better. First, too many teams tried too many experiments and came up with too many possible solutions, none of which fitted together. Six months into the initiative—their original deadline—they were no closer to having a working cloud native platform than they had been at the start.

Again, they tried to adjust. They tried to clean up the too-many-platforms mess by calling in a systems architect (with zero experience in Kubernetes) to design a unified approach. He came up with an impressive diagram that, unfortunately, got it all wrong. Even more unfortunately, there was still not enough understanding of how cloud native works, within WealthGrid’s ranks—from corner office to middle management to engineers—for anyone to see how wrong it was. Thus everyone set to work trying to implement the new but still nonfunctional architecture. Again, everyone worked hard. Six more months passed. Again, nothing was delivered.

Meanwhile, the skeleton crew of engineers maintaining the original system was able to keep everything working, but they weren’t able to deliver new features. The team was understaffed. And besides, why invest in new functionality on the old system when it was about to be replaced? Initially nobody worried about this, because the new cloud native system would have such velocity that WealthGrid’s feature debt would surely be paid back in very short time. But then the promised six-month delivery time turned into a year, and still the new platform was nowhere near ready.

This meant a year of no new features or other meaningful improvements for WealthGrid’s customers—and a very real risk of losing market share. The sales and marketing teams got the ear of the CFO, who talked to the CEO and the board, who finally laid down an ultimatum: Here are five new features that we need, and fast. We don’t care about which platform we use. *Just deliver them*.

Now what does WealthGrid do?

**PHASE 1: THINK**

Phase 1 of the transformation design is called “Think,” because it’s all about ideas, strategy, and objectives.

Most companies are going to enter this phase at a very similar standpoint: they are using Waterfall to Agile-ish delivery methods focused on very high proficiency. This means that most of the talent and resources in the company are dedicated to delivering a very stable core business product or service. There might be a bit of experimenting going on here and there, but it is definitely not an internal priority. There’s probably no team dedicated to innovation, or any programs designed to inject creativity into the existing system. In short, this is very likely a company that has forgotten how to be creative.

To put some numbers on this, at this point the distribution of investment between delivery, innovation, and research is 95/5/0. The vast majority of focus, 95%, is on delivering the core product as efficiently as possible. A small amount, 5%, is left over for any innovation going on. Whatever this is, it’s targeted mainly at improving systems around the core product, and expected to be useful or pay off very soon. Zero research is going into any kind of crazy ideas that might—or might not—pay off big someday.

It is against this backdrop that the first stirrings toward transformation take place. Our first set of patterns addresses getting the process started: first, effective strategic thinking and decision making from the organization’s leaders. And then converting this strategy into vision and objectives to move the process to the next phase, execution.

**Enter the Champion**

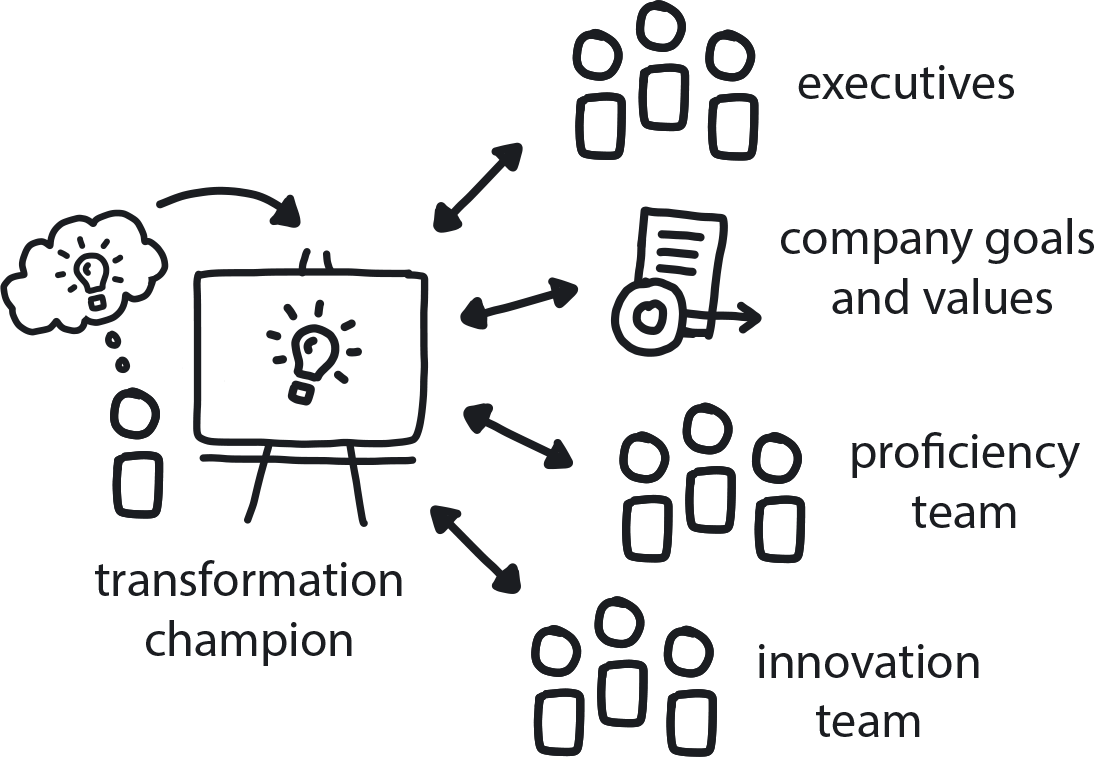
Any significant innovation always begins in the same place, or rather with the same person: the Transformation Champion.

A large, established system in motion is usually slow to change direction: see Newton’s First Law of Physics, also known as the law of inertia. And WealthGrid, like any other sizable and successful company, has invested a lot of time—years, if not decades—in streamlining its core business processes to be as proficient as possible.

The problem with achieving this impressively high proficiency, as we saw in [Chapter 7](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#patterns_for_strategy_and_risk), tends to be a corresponding decline in the ability to innovate. Every successful company was once a startup, and startups are all about creativity. Its ultimate goal, though, is to get to proficient, even algorithmic, delivery of their product or service, because that is where the profits lie. Once they make it, successful enterprises typically focus almost exclusively on operating as lean as possible. This is fine, good even, except that when they focus entirely on proficiency, they forget to keep a piece of creativity alive.

So when any company similar to WealthGrid decides to change course—i.e., innovate—even when it’s in response to an undeniable threat to its bottom line, it must overcome a fair amount of resistance. Just as a catalyst like a disruptive stranger coming to your town is needed to stimulate the change, a corresponding force—a metaphorical town mayor—is needed to manage the community’s response. In a cloud native transformation, this person is the Transformation Champion and the whole thing starts with them. There has to be trust in their leadership and willingness to follow their guidance.

**PATTERN: TRANSFORMATION CHAMPION**



**Figure 11-1. Transformation Champion**

The Transformation Champion is a person (or sometimes a small team) who understands both the transformation and the company objectives, is well connected within the organization, and is highly motivated to promote the transformation. Unless they are recognized by the company’s leaders and given authority, however, they will be unable to stimulate effective change across the organization.

Every company has some of these people. They see the future more clearly than most, and as a result they always want to change something. This is the hallmark of a true champion: they don’t just have ideas; they try to take action. Champions are good at promoting a new idea while making sure it is a proper fit for the company’s goals and values. Such people need to be identified, nurtured, and trained in general, but when it comes time to undertake a major initiative that will terraform the entire organization, they are critical for its success.

The trick is recognizing the champions in your midst, because this first pattern is not about *creating* a Transformation Champion—it’s about *discovering* the one(s) you already have. These people tend to be self-selecting, and quite often they are the trigger for the whole thing.

So, look around! In every company we visit to consult about cloud native, there is one person super excited we are there. This is often the person who asked us to come in the first place.

Once we take on the client, our first question is, will that person lead the initiative, or will they name a different leader? This can be tricky. Not everyone is cut out to manage a large transition project. But in general this self-selected evangelist is going to be the right person to lead yours, and so it’s important to appoint someone else only if there is a truly compelling reason to do so. (Said compelling reasons, FYI, do not include “In our company senior managers always lead large projects.” Trust us on this. You’re going cloud native now and it’s time to do things differently.)

If you seek your champion, you will find them. Somebody inspired this whole transformation thing, and this person could be anyone anywhere in the company, engineer or manager or CEO. Finding this person is not the problem—they exist. The true problem is that, usually, they just don’t have the power to get things done right. Fortunately, the simple fix is simply to empower them and then get out of the way!

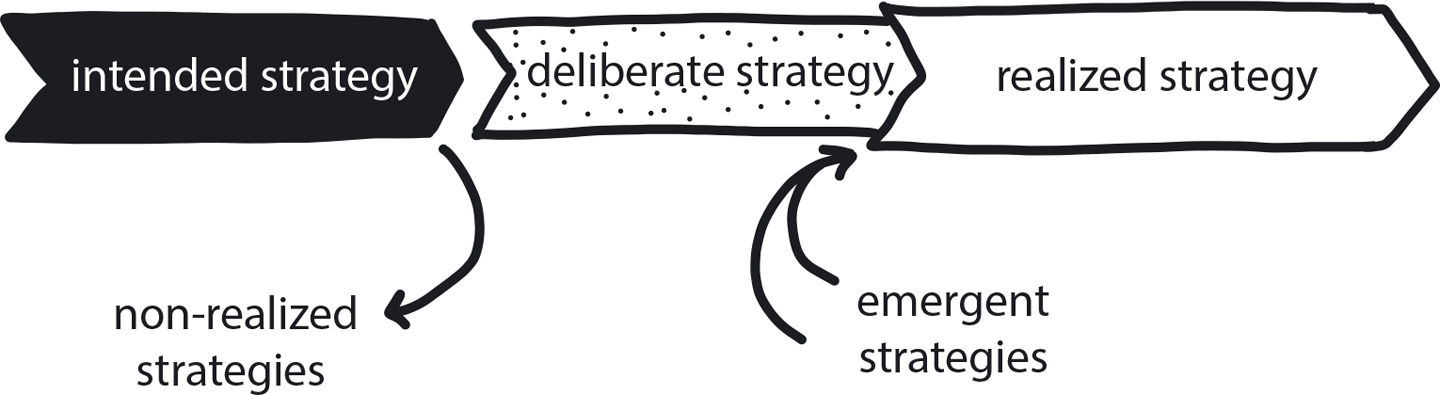
So, once you find them and anoint them as the authorized Transition Champion, let this be known far and wide. Send a clear message that this transformation is an initiative supported by the company—not just a side project by a few motivated employees—and that participation is not only appreciated but expected. This is absolutely crucial. A champion can tell everyone the absolutely right thing to do as their part of the migration, but unless they are fully supported by the executive team (or are themselves very senior) people can ignore them and do nothing, or even block their efforts.

WealthGrid’s Transformation Champion is Jenny, by the way. She is a classic example: Jenny saw what her company needed to do in order to be ready for the ways the market is permanently changing. And then she stepped up to make it happen.

You may rightfully wonder why, if Transformation Champion is an essential pattern and Jenny is a good example, did WealthGrid’s efforts then fail not once but three times? After all, Jenny knew what needed to happen: the company had to go cloud native in order to remain competitive. She even, eventually, had the full confidence and backing of the CEO and board to do a full-on initiative, not to mention hands-on help from much of the company’s engineering staff. What was missing?

The missing piece was Dynamic Strategy.

**PATTERN: DYNAMIC STRATEGY**



**Figure 11-2. Dynamic Strategy**

Today’s tech-driven marketplace is a constantly shifting environment, no matter what business you are in—so your game plan needs to shift right along with it.

WealthGrid’s first transformation attempt, with Jenny’s team trying to do it as a side project, did not have any real strategy at all. In the second attempt, though, everyone got on board, a strategy was created, and things still went sideways. Had the Dynamic Strategy pattern been applied, however, the story would have made the same missteps, only much more quickly. Instead of wasting a year to figure that things were simply not working, they could have moved through the process in two months. It would still have been painful, but at least it would have been quick.

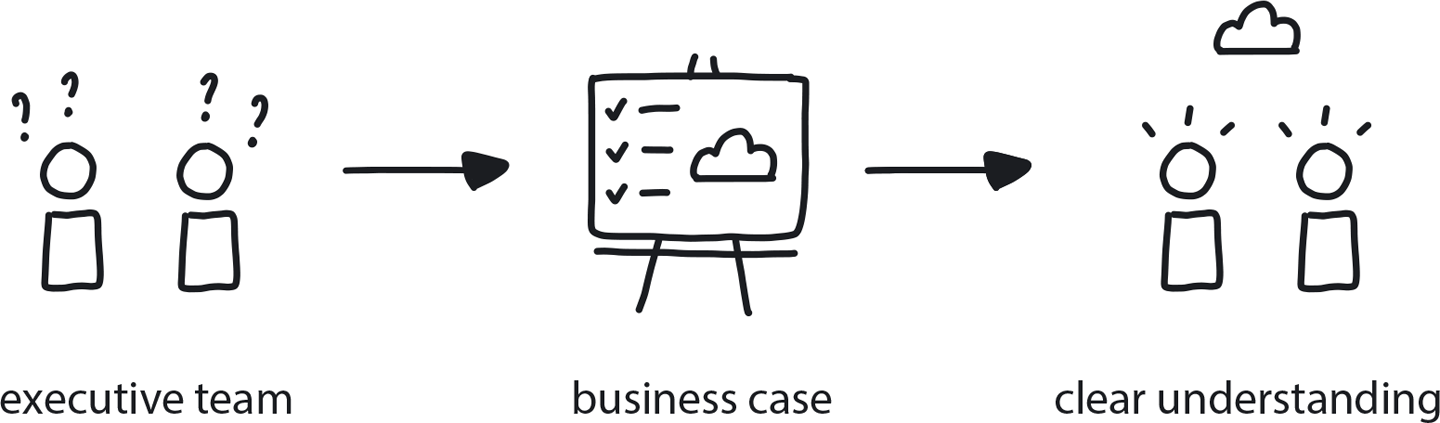
Dynamic Strategy is essential for doing cloud native right, from inception to completion. This is actually a super pattern, one so essential that it overlays all the other patterns in our transformation design from the very beginning, and stays forever. Let’s look at the next set of patterns now to learn more specific steps for getting a transformation off to the right start.

**Ready to Commit**

Even though it’s time for enterprises to learn how to navigate this new paradigm, we aren’t saying to forget the past. We learned a lot from WealthGrid’s experience thus far, and we need to use it. The first lesson being, don’t continue something from the past just because your organization started it. That is classic sunk cost fallacy, or irrational escalation: the phenomenon where people justify increased investment in a decision, based on the cumulative prior investment, despite new evidence suggesting that the decision was probably wrong. This unfortunately is a very frequent phenomenon in companies moving to cloud native. People routinely push forward with projects that are obviously not going to bring any value, just because they’ve already invested so much in the mess.

So let’s use Dynamic Strategy to pause and reevaluate here at the very start, and make sure we are not going into sunk cost. Now is the time to ask, *Do we even need this thing?* using the Business Case pattern.

**PATTERN: BUSINESS CASE**



**Figure 11-3. Business Case**

When an organization’s executive leadership does not fully comprehend the advantages that result from a cloud native migration, a strong Business Case allows them to understand and support the project without hesitation.

Sometimes Business Case can be a tough sell even when a cloud native transformation is clearly the right response to an existential threat. The traditional model is for organizations to be massively risk averse, which means minimizing uncertainty at all costs. This is why it takes forever to plan anything in Waterfall—the process is designed to identify and mitigate every conceivable problem in advance. So pitching a cloud migration to a change-averse culture that avoids new technologies or experimental approaches can be challenging. Cloud native is complex, and the benefits are not easily visible, especially at first. So, for all these reasons, some companies are simply going to avoid talking about the risks and advantages. In these situations, Business Case facilitates the conversation in a factual, objective, and non-threatening way.

For most, though, Business Case is a valuable reality check before diving into an initiative they are already inclined to take on. Too many organizations jump on the cloud native bandwagon without doing a true cost/benefit analysis. (In fact, the cognitive bias known as the “bandwagon effect” exactly describes how companies caught in the hype of the cloud conversation make decisions without understanding exactly how a transformation fits with their business needs and goals.) Thus, evaluating the Business Case should involve business stakeholders, information-gathering interviews with internal and external subject matter experts, and a clear vision for where the organization is headed.

This may reveal that going cloud native is not actually the right thing for a company to do, at least not right now. There are a few times when cloud native likely would not be the right move, and an organization working through Business Case needs to be aware of them even if they’re uncommon.

* When a company is struggling for survival and has no budget or time to allocate to the transformation.
* If there is an immediate crisis that cannot be solved with cloud native. This is a case of “right move, just not now.” Cloud native will have no appropriate priority in this case, but once the situation has resolved, Dynamic Strategy tells us to revisit the decision.
* Maybe in a very stable market where nothing changes. But these markets are very few, and becoming fewer all the time, so this is a high-risk decision.

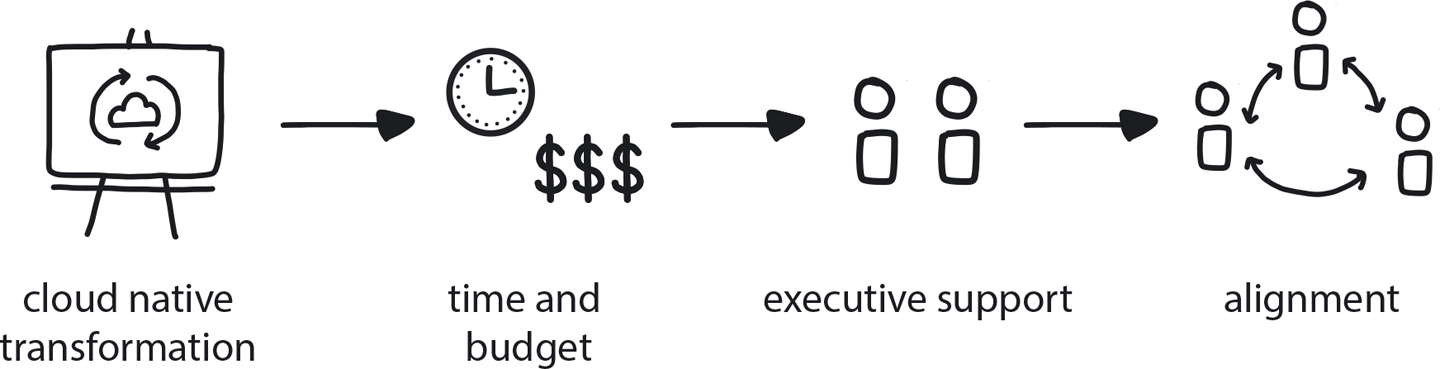
The best reason not to do cloud native at all is when the software part of the business is really simple and limited, and cloud native will overcomplicate things without adding significant value. But this mainly applies to smaller companies, since any large enterprise should have substantial IT that will benefit from a migration.

Everyone else should start investing in cloud native. The only question is pace: some can do it slowly, while others—for example, anyone in a stranger-danger situation—should do it with higher urgency.

Keep in mind the difference between cloud computing and cloud native. Namely, the cloud refers to on-demand delivery of virtual infrastructure (hardware/servers), storage, databases, etc., while cloud native is an architectural approach—a methodology. Even monoliths can benefit from taking advantage of the automation, ability to pay only for resources consumed, and other niceties built into public cloud platforms. Any enterprise running a highly stable monolith on a legacy codebase that requires very little change or updating ever needs to take a very hard look at the Business Case for regrooming that monolith into microservices. (Don’t stop reading, though. There are many more patterns to help you move your functional monolith onto the cloud!)

Business Case in hand, the next step in our transformation design is the Executive Commitment pattern.

**PATTERN: EXECUTIVE COMMITMENT**



**Figure 11-4. Executive Commitment**

To ensure allocation of sufficient resources and reasonable delivery time frames, large-scale projects such as a cloud native transformation require strong Executive Commitment.

This function of this pattern is to establish the cloud native transformation as a high-priority strategic initiative with explicit support from the company’s executive management. Public announcement of the cloud native transformation as a key strategic initiative creates company-wide alignment and awareness, while also setting the expectation of collaboration from all departments within the organization.

It might seem that Executive Commitment for a transformation should go without saying, but it’s an important moment in a migration. It establishes that this is not some small engineering project driven by a handful of motivated employees but a full-company effort, and the senior leaders are standing behind it. This is needed for a number of reasons. First and most obvious is that commitment from the company’s most senior leaders makes sure the project gets an appropriate level of resources and budget allocated. Less obvious but equally important is that announcing their full support makes it known across the organization that change is coming, and that the cloud native transformation is an official part of the company’s value hierarchy. Maybe the project doesn’t need to be *the* top priority, but it is indeed a real one.

WealthGrid’s first attempt failed in part due to a lack of Executive Commitment. Though Jenny got permission from upper management, she didn’t get support. No extra budget or resources were given to the project, and it remained a small, localized effort that most people in the company had no idea was even happening. This is exactly the reason why it all gets spelled out as a pattern: Such commitment from the management needs to include preparation of a Transformation Strategy (pattern coming up), public announcement of the project, and the allocation of adequate resources and budget. Again, had Jenny gotten full Executive Commitment for her first try, it would have gone quite differently. It may still have failed, but it would have failed much faster and according to an actual strategy—which could then be refactored for lessons learned.

When an organization stands poised at the first step of a migration journey, it is important that the leaders, well, lead the way. Because when the cloud native destination is reached, the company will not be the same company. It will build in a different way, compete in a different way, generally behave in a completely new way. And because change is scary, the CEO and board need to lead the way with confidence.

That is why the moment where Executive Commitment is achieved is the moment a transformation truly begins.

**Vision and Core Team**

Having named a Transformation Champion (fortunately, WealthGrid was smart enough to officially name Jenny to the role she had been filling all along), established the compelling need for cloud native with Business Case, and achieved Executive Commitment for making it actually happen, it’s now time to plan the actual transition.

Two things need to happen now: we need to create a proper migration strategy and pick the team to deliver it. In its first attempt to go cloud native, WealthGrid had neither. Not on purpose, but because it simply made the same mistake most companies do when launching a transformation: simply adding the transformation to the usual Agile backlog of tasks. Functionally, this means tasking an existing team with building the new system while still being responsible for their regular duties, too. Jenny’s team never had the chance to build much of anything at all. But even when a team does get more time to work on it than they did, or has simply a long timeframe to do it in, they still won’t be able to deliver. Limited experience coupled with a lack of space and flexibility for research leads to pursuing cloud native implementation using “well-known ways”—that is, trying to deliver cloud native using incompatible approaches from Waterfall, Agile, or a combination of the two. As we have seen, the initiative falters.

Two cognitive bias factors influencing things here are the ambiguity effect, where people facing a situation with a lot of unknowns will fall back on simply doing what they already know how to do, using methods that have worked in the past. Similarly, law of the instrument leads them to choose tools that they know, rather than seeking the right ones for the job. When there is time pressure and little or no opportunity to explore new tools and techniques, guess what is going to happen? Yes, people will pick up their comfortable tools without even thinking.

Furthermore, without an overall consistent vision and a core team in charge of delivering it, different teams will make independent and frequently conflicting architectural decisions. This is exactly what we saw happen with WealthGrid’s second attempt.

To avoid both of these problems, we use the Vision First, Objective Setting, and Core Team patterns to set the correct path right away. These mark a gradual shift away from abstract strategy toward concrete execution, which moves down the chain to middle management and the tech teams. But first, before we leave Steve and the rest of WealthGrid’s C-suite execs, we have one more pattern for them: Decide Closest to the Action.

**Delegating Power**

Traditional organizations are all about delegating responsibility, but not authority. Important execution decisions are rarely if ever delegated to those actually in charge of carrying them out. This pattern is about putting the decision close to where the change is happening, and doing this is key to successfully evolving to a cloud native culture.

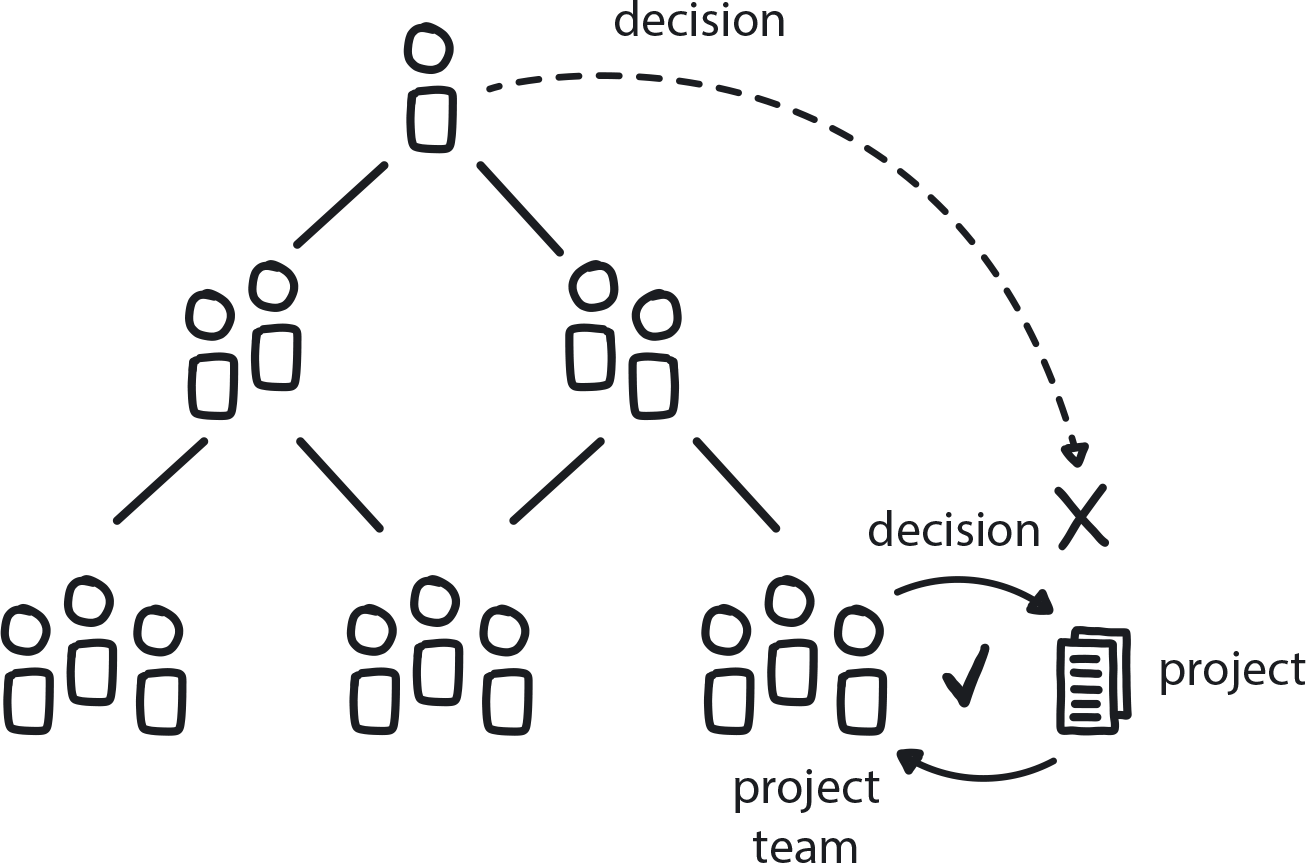
In the past, the flow went: strategy -> vision -> execution. All planning happened at the first step and almost always at the top of the chain of hierarchy. Once you moved on to vision, planning was done; strategy was set and execution was to follow. Static strategy doesn’t work in cloud native, as we have seen with the Dynamic Strategy pattern.

In a way, the Decide Closest to the Action pattern is about dynamic planning and delegation. The simple (but not easy) idea behind it is for decisions to be made at the point of action by the person who’s going to be doing the action. So, if your team is responsible for delivering a particular microservice, it is your responsibility to do the planning around that microservice. Not your boss, not the project manager, not the systems architect, certainly not the folks several flights up in the nice offices. *You.*

This is where this pattern perhaps gets a bit tricky to implement. In traditional organizations, managers typically do the vast majority of planning and decision making. Then they hand down a set of pre-specified tasks to the engineering teams to execute. Cloud native velocity, however, depends on teams being able to make fast decisions regarding whatever task they are working on, without needing to consult the boss. The managers have to be willing to give up their traditional position of control in order for this to happen.

This is important so we will say it again: in cloud native, the engineers get to plan and to make decisions regarding what they’re working on. They don’t get handed a set of marching orders from a project manager to fulfill exactly as specified. They don’t need to seek permission to try something different if they spot a better way to go. In cloud native, the power to decide rests with the person doing the job.

**PATTERN: DECIDE CLOSEST TO THE ACTION**



**Figure 11-5. Decide Closest to the Action**

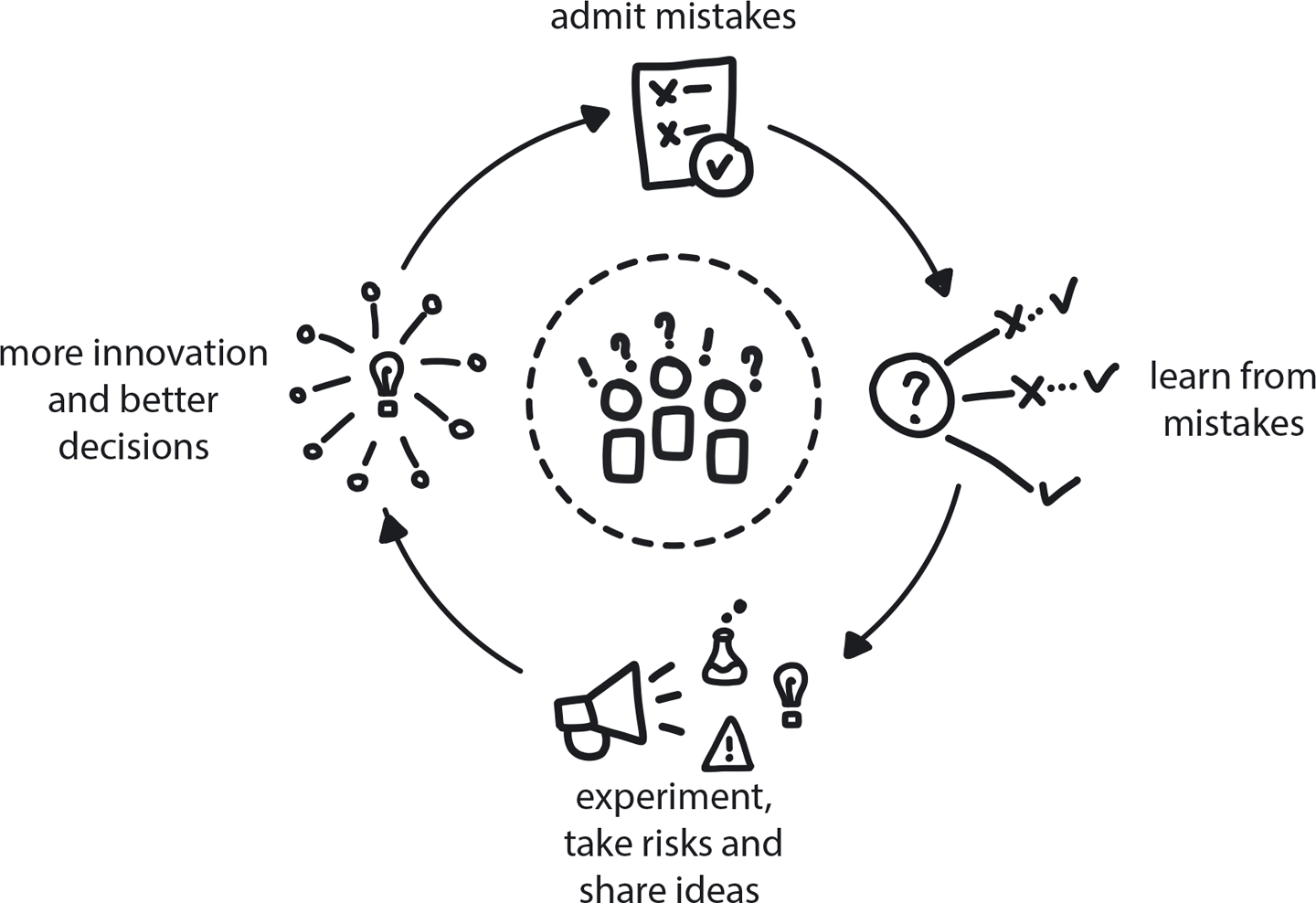
Those closest to a change action get the first chance to make any decisions related to it.

Like Dynamic Strategy, Decide Closest to the Action is a super pattern that applies not just in every phase of the transformation but permanently across the entire organization. Paired closely with Decide Closest to the Action is another super pattern, Psychological Safety. This addresses the need for people throughout the organization to feel they can express ideas, concerns, or mistakes without being punished or humiliated for speaking up.

In a work environment that lacks Psychological Safety, team members hesitate to offer new or different ideas, especially when these may diverge from the overall group opinion, and will generally act in risk-averse ways. Since cloud native depends on teams collaborating to come up with innovative solutions through independent, divergent thinking, Psychological Safety is essential for creating an environment that makes this kind of co-creative work possible.

If teams lack the ability to fail safely when experimenting, they waste time exhaustively analyzing all future outcomes to minimize exposing themselves to risk, effectively killing both creativity and velocity. In a psychologically safe work environment, though, people are willing and even excited to try, possibly fail, and then try again as they strive to come up with the next idea (which might be wild, but also possibly game-changing). And they will be able to make decisions with confidence, knowing that their team will tell them—honestly yet constructively—when they might be making the wrong one.

**PATTERN: PSYCHOLOGICAL SAFETY**

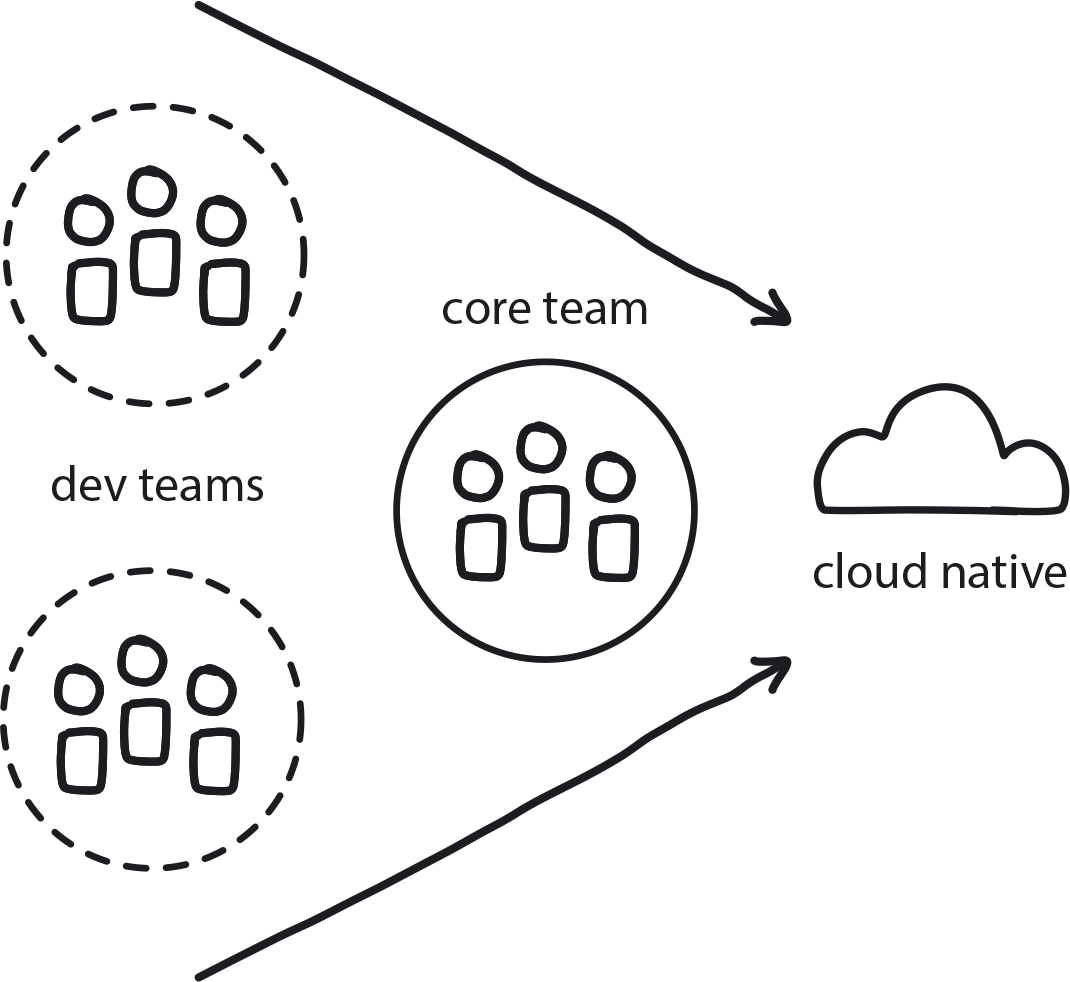


**Figure 11-6. Psychological Safety**

When team members feel they can speak up, express concern, and make mistakes without facing punishment or ridicule, they are able to think freely and creatively, and to be open to taking risks.

Once these innovation-empowering cultural patterns are put in place, the company is now ready to take its first concrete step toward the cloud native transformation: naming the Core Team.

**PATTERN: CORE TEAM**



**Figure 11-7. Core Team**

A dedicated team of engineers and architects, constantly diving deeper into technical challenges, will reduce the risk of the transformation. The team members gain experience that will help onboard the remaining teams more quickly and effectively.

Creating the Core Team means handpicking a relatively small team to drive the overall innovation and, especially, take charge of the initial stages.

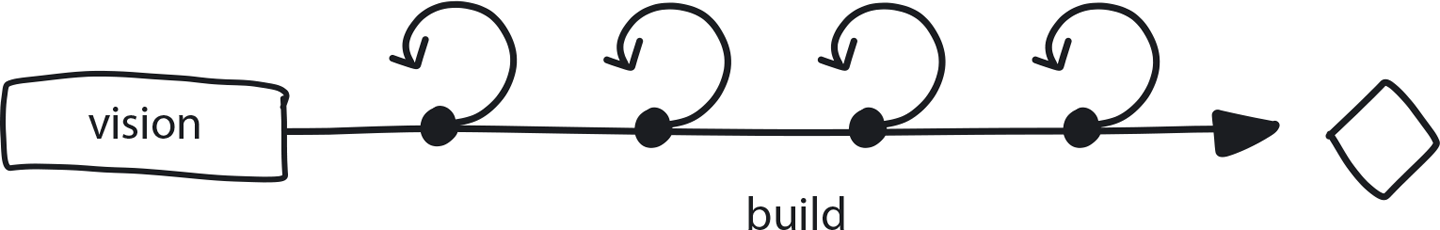
What does a Core Team look like? Well, it’s compact—typically, five to eight engineers (including software architects, if you’ve got them to draw from). Team responsibilities will include ownership of the technical vision and architecture, de-risking the transformation by running a series of Proofs of Concepts (PoCs), creation of a Minimum Viable Product (MVP) version of the platform, and later onboarding and guiding other teams. (No worries, we’ve got patterns coming for all these.)

Essentially, the Core Team rapidly iterates through the most challenging parts of the transformation. They begin by establishing a high-level vision and then translating the vision into concrete, executable objectives. They use research and experimentation to build their own knowledge and experience in the cloud native paradigm—using the knowledge they gain to adjust the transformation vision and architecture as they go. Later, the Core Team’s first-hand understanding helps them to onboard other teams to the new way of working. Their knowledge paves the path for the rest of the company for a seamless and successful cloud native adoption.

The team often stays together even once the transformation is done, because they have effectively become the main repository for the organization’s cloud native knowledge. Sometimes the Core Team, or perhaps a portion of the team, forms the Platform Team (more on that soon). Or, once the new system is running fully in production mode, they may remain in charge of continually improving the platform—remaining in an innovation-and-research mode to help the company stay ready for whatever future comes next.

Once chosen, the Core Team’s very first task is Vision First, outlining a high-level transformation plan.

**PATTERN: VISION FIRST**



**Figure 11-8. Vision First**

Defining a guiding vision as very first step helps set the right path through an uncertain environment.

In Vision First, the Core Team needs to define a clear and achievable vision that can be translated into specific executable steps. This vision defines and visualizes the architecture of the whole system up front. Since the Core Team is likely still themselves early in the cloud native learning curve, this vision can be created with the help of external resources, such as consultants. This will definitely help the vision stage move quickly.

If time allows, the Core Team can work to find their own way, uncovering the vision/design through a series of small research and prototyping projects, ranging from a few hours to a few days each. This DIY approach is ideal, since it hastens the learning that the Core Team needs to undergo anyway, but it can make this initial phase of the transformation take longer to complete. An experienced consultant can help jump-start that learning, though, while steering things in the right direction, so perhaps a combination of the two is the best of both worlds.

No matter which way you get there, Vision First outlines the technical and organizational roadmap for the company for the next year or two. As the team members work to create this vision, they need to remember that this is not carved in stone—vision needs to be dynamic, just like strategy. At this point, they don’t have the full picture of what they are trying to achieve and the initial vision is just their best guess at the very beginning. Don’t get stuck on this early idea, because it needs to evolve as the transformation progresses and new information is uncovered.

The question to answer is, essentially, *Where are we going?* It’s important to keep the answers high level enough to allow freedom of choice during implementation (in other words, don’t start choosing tools just yet). At the same time, though, things need to be detailed enough to provide clear guidance, which will help avoid common pitfalls.

The practical implementation of creating a cloud native transformation vision is, *do* name names—of architectural elements. *Don’t* start choosing specific tools or service providers at this stage.

*Good things to include*

Semi-technical terms describing broad concepts, such as “microservices” and “containerized platform in the cloud.”

*Things to avoid:*

Committing to any specific tool or tech, no matter what kind of great things you’ve heard about it. (Don’t worry, we will be getting to that very soon, and by way of applying patterns to help make sure your choices are the right ones.)

Once the Core Team lays out the Vision First pattern, it moves quickly and directly to translate it into concrete objectives. This is where we begin to create specific architecture and a series of steps to move toward the Maturity Matrix cloud native standard (see [Chapter 5](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#knowing_thyself_the_cloud_native)).

**PATTERN: OBJECTIVE SETTING**



**Figure 11-9. Objective Setting**

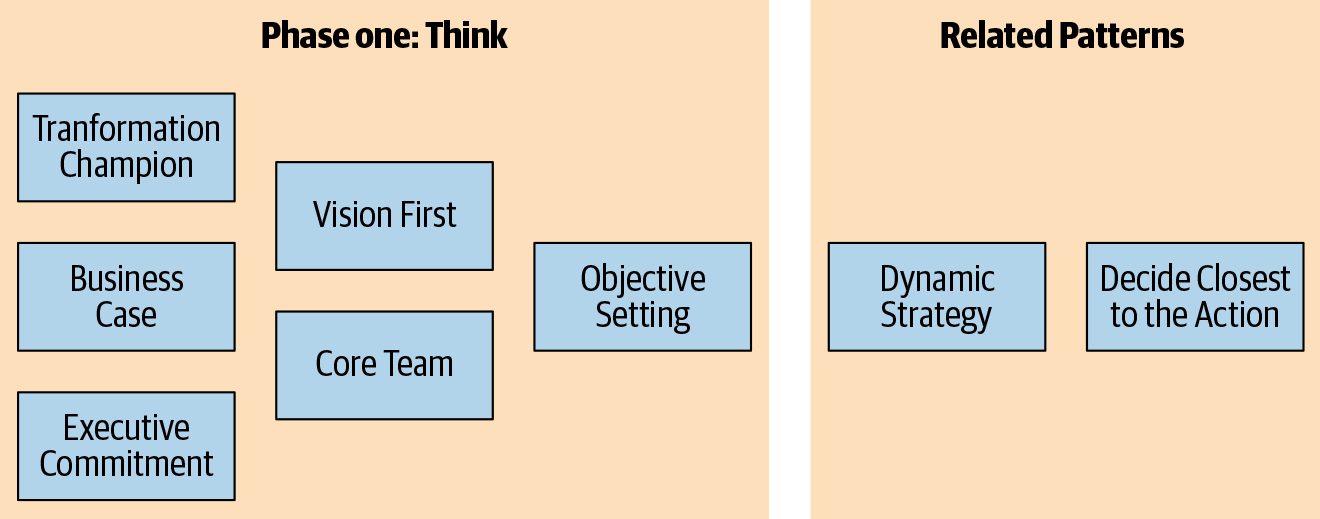
After establishing a transformation vision, the next step is to translate it into pragmatic goals and actions for moving the initiative ahead.

The Objective Setting pattern marks a pivotal point, because this is where we move from theoretical strategic planning to defining concrete, pragmatic, and above all executable steps toward the cloud native goal line.

The whole thing began with a strategic, executive decision-making phase, which—aside from the Transformation Champion, who can come from any part of the organization—was chiefly in the hands of the company’s senior leaders, CEO, and board. The journey there was abstract, moving through establishing a case for and then committing to a transformation initiative. The key outcome was to make a firm decision: *We want to be cloud native and we are willing to provide the resources to make it happen.*

The Core Team was then created to begin turning strategy into reality. First, defining the vision—the high-level roadmap for the transformation—and then translating that into an architecture and a set of achievable objectives: *We want DevOps teams structure, need to refactor into containers and microservices, and want to install an orchestration platform to manage everything.*

The time has come for creating the steps that will pragmatically achieve these objectives: A framework of tasks to achieve specific goals—first we need a basic platform, so we will do experiments, try some PoC—that becomes execution. [Figure 11-10](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch11.html#phase_1_Think_the_first_step) shows the patterns that we’ve followed thus far.



**Figure 11-10. Phase 1: Think, the first step in our cloud native transformation design, covers strategy and objective setting.**

Thus have we progressed from idea to strategy to objectives, which brings us to the end of Phase 1. It’s time to move into Phase 2: Design, where we begin to identify the elements for constructing our cloud native platform.

**PHASE 2: DESIGN**

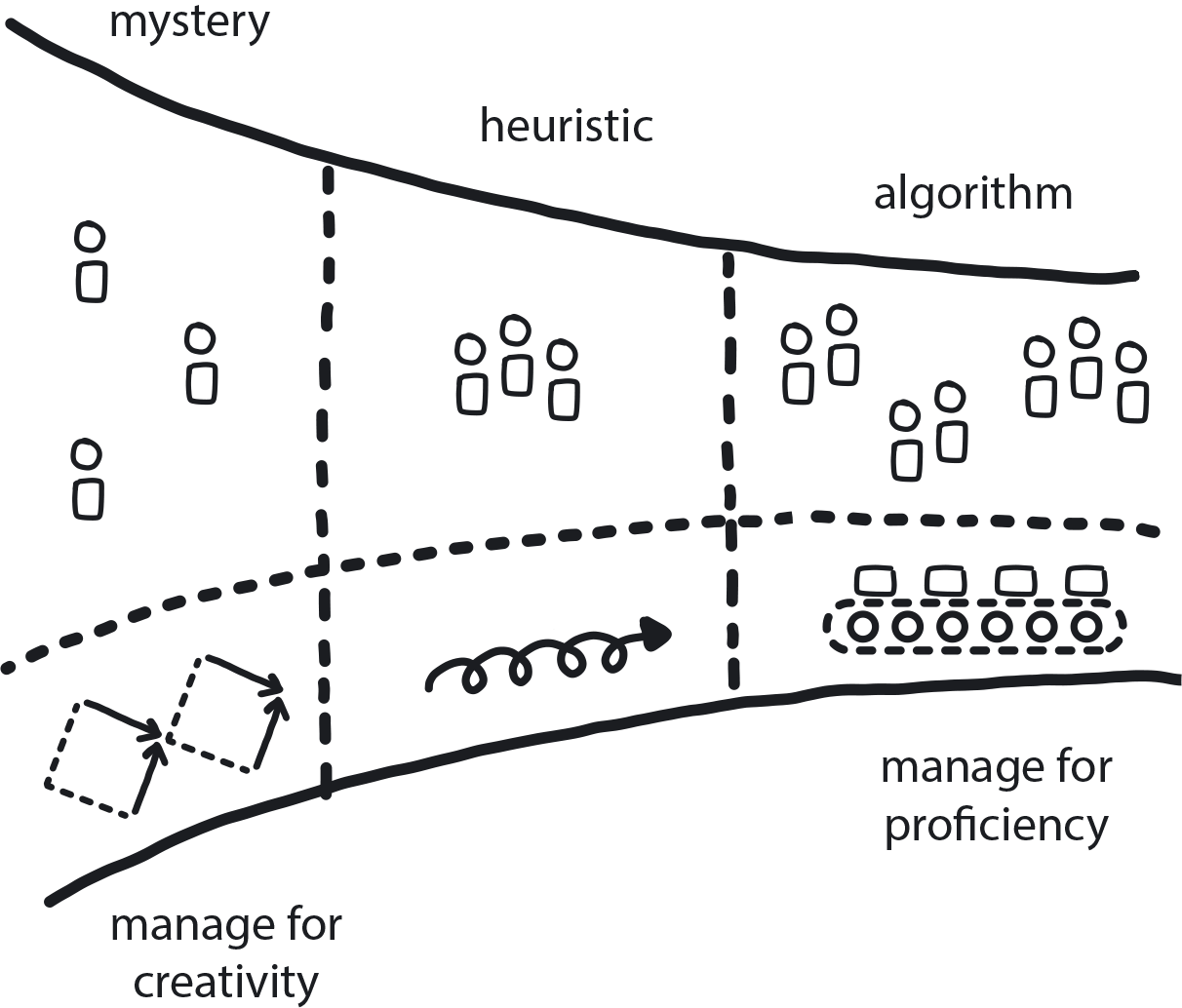
Phase 1 gave us a first, rudimentary roadmap for finding our way to the cloud: a high-level vision of Microservices Architecture delivering Containerized Apps in the cloud by Build-Run Teams. Wait, what are all those things?! Don’t panic: these are all patterns that, along with others, will carry us from vision to delivery. Let’s take a look.

The pivot point between Phase 1 and Phase 2 consists of splitting the organization, at least temporarily. The Core Team is setting off to design and build the new platform, while the rest of the company carries on with business as usual for the time being.

These two paths require separate sets of patterns and separate approaches. The Core Team is managed as a creative team, given a purpose and the freedom to experiment and research their way to answers. The rest of the teams still keep building the value in the existing system, until the new platform is ready. They are managed for proficiency.

Not to give short shrift to the proficiency teams—they are busy making money for the company while the Core Team innovates—but they only get one pattern for the time being. Most teams will eventually need to evolve to cloud native’s more flexible, intuitive, and problem-solving way of working. During the transition, however, it’s important to keep the proficient teams proficient and run them as they have always been run, not changing to cloud native techniques yet. Since they aren’t changing much (yet—their time is coming soon!) the only thing to discuss right now is how best to manage them in the interim.

**PATTERN: MANAGE FOR PROFICIENCY**

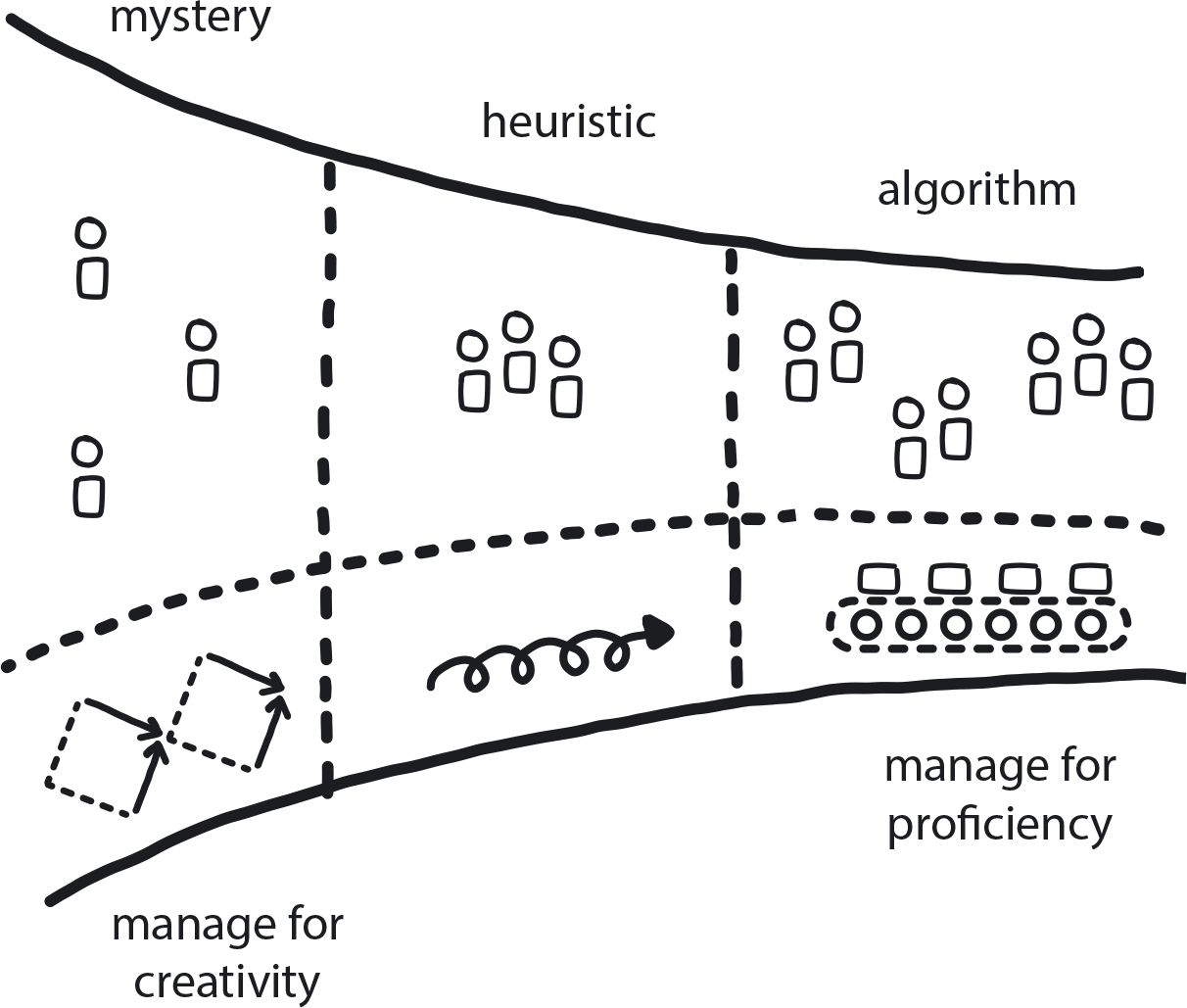


**Figure 11-11. Manage for Proficiency**

Teams delivering stable and highly repetitive or algorithmic work should be managed for high quality and optimal efficiency.

We have named this the Design phase, but it could just as easily be called the Exploration or Innovation phase. Now is the time that the Core Team delves into creative research, and they need to be appropriately managed for producing innovation.

**PATTERN: MANAGE FOR CREATIVITY**



**Figure 11-12. Manage for Creativity**

Teams charged with innovation need the open-ended freedom to experiment their way to solutions without pressure for delivering specific results on a set schedule—and the freedom to sometimes fail along the way.

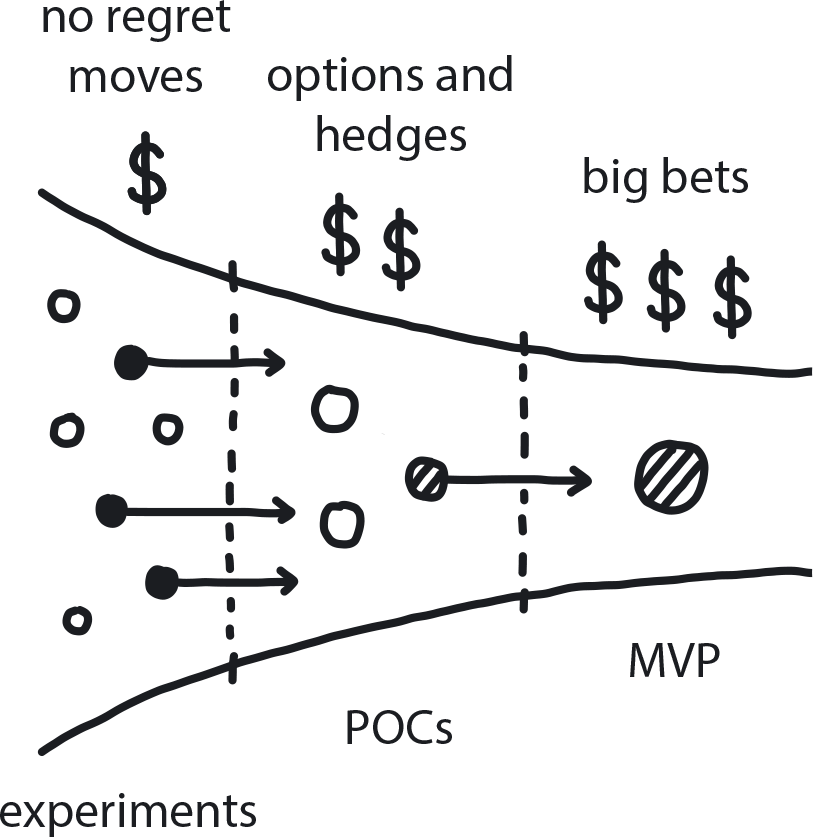
In their first attempt to go cloud native, WealthGrid made the extremely common mistake of having an existing team work on building the new system while still responsible for their regular duties, too. The obvious problem is that urgent issues on the existing system—the one that is making money for the company—will always take priority over building something new that isn’t going to add value for who knows how long. For that reason it makes sense that the Core Team should do nothing but focus on the transformation. It also makes sense that this team is going to need some intense time investment for learning and experimentation, so they shouldn’t be distracted by unrelated work.

The most important reason that the Core Team needs to stand alone, however, is because they are breaking away from a longstanding way of doing things in order to do something completely new and different. This means that the team needs to work, and be managed in their work, in a completely new and different way.

Remember [Chapter 7](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch07.html#patterns_for_strategy_and_risk) and all that talk about proficiency vs. creativity? The Core Team needs open-ended time to work, and this can be a challenge for others in the organization. Managers are used to overseeing responsibilities you can, well, manage, in a predictable, top-down, hands-on way. Managing for creativity is more about facilitating innovation: providing the opportunity for innovation to incubate by defining the goal, providing the proper circumstances, and then stepping back. You cannot manage facilitation.

Practically speaking, how does the Core Team start to make smart design choices for this new cloud native system while minimizing risk? Gradually Raising the Stakes.

**PATTERN: GRADUALLY RAISING THE STAKES**



**Figure 11-13. Gradually Raising the Stakes**

In an uncertain environment, slowly increase investment into learning and information-gathering actions; eventually you uncover enough information to reduce risk and make better-informed decisions.

Related sub-patterns: No Regret Moves, Options and Hedges, Big Bet

In the predictive world of Waterfall, when you launch a big project, you can typically make big decisions right away. The project itself might be new, but it’s almost certainly going to be a lot like previous initiatives. Now, however, the Core Team is feeling their way onto the cloud with only a very basic map. The next move is far from clear because they are still early in the process of building their own cloud native knowledge, and low-certainty projects carry extremely high risk. Any decision made early on in this uncertain environment is highly likely to turn out to have been a wrong choice.

Gradually Raising the Stakes is a super pattern and a general strategy for cloud native risk management that is critically applicable to this pivotal point in a transformation. This pattern gives us the tools to address uncertainty by moving step by logical next step. The way it works is through three graduated levels of risk taking: little risk, moderate risk, high risk. In terms of pure strategy formation these are the sub-patterns No Regret Moves, Options & Hedges, and Big Bet. There is a lot more about the Gradually Raising the Stakes super pattern and its sub-patterns in [Chapter 5](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch05.html#knowing_thyself_the_cloud_native).

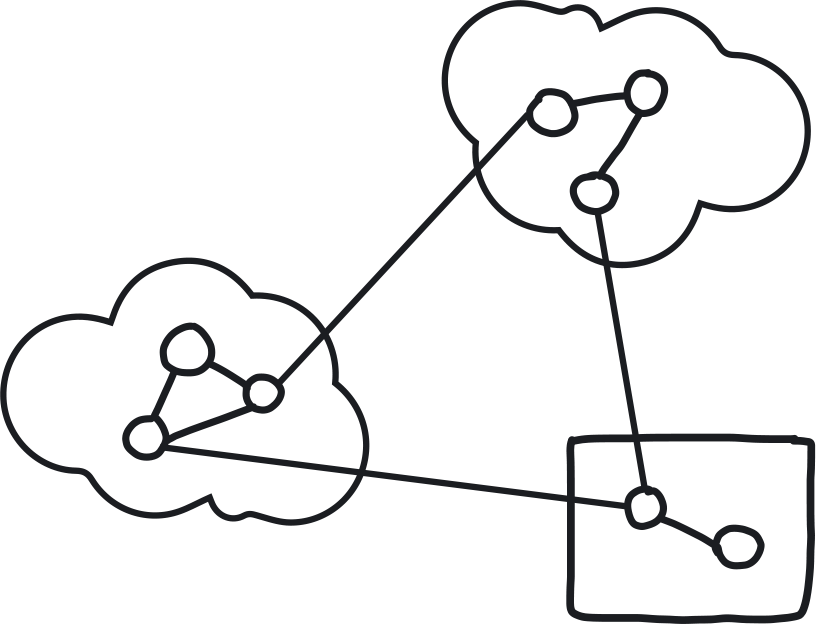
When it comes to creating a transformation design, however, these patterns correlate with three technical patterns. Applied in proper order, these are practical steps for reducing transformation risk: Exploratory Experiments, PoCs (Proof of Concepts), and Platform MVP (Minimum Viable Product). We will encounter these patterns now, along with other patterns that are important at this stage of the initiative. These are mainly around adopting new, cloud native-oriented ways of thinking and doing things.

The gateway to these is the Distributed Systems super pattern, because the rest of the transformation is essentially implementing an optimized cloud native version of distributed systems architecture. The Exploratory Experiments pattern, to establish the initial framework for our cloud native explorations. We then unpack Distributed Systems super pattern and all the related pattern puzzle pieces the Core Team will get to play with.

**Distributed Systems and Friends**

Welcome back to the most technical subject matter covered in this book: the broad technologies and methodologies that together compose cloud native architecture. The Core Team is experimenting to find the best implementation of each pattern for their particular transformation initiative—a level of technical detail we are saving for the sequel to this book—but the pillars are unchanging. It all begins with Distributed Systems.

**PATTERN: DISTRIBUTED SYSTEMS**



**Figure 11-14. Distributed Systems**

When software is built as a series of fully independent, loosely coupled services, the resulting system is, by design, very fast, resilient, and highly scalable.

The development of a monolithic software system is limited by its sheer complexity, which only a few top-level architects and managers may be able to comprehend (but even this is not guaranteed). Indeed, it is tempting to start a new software project as a monolith to get things started quickly, since the architecture is simple, and the few components share resources in a single codebase.

As the system expands, however, the size and complexity of its major components keep growing. Over time, this causes a variety of bottlenecks to develop, mainly in the form of dependencies on all different levels: data sharing, calling functions from other components, and teams being held up by other teams. All this leads to an increasing level of coordination required to produce any change, to the point where the overhead of coordination becomes larger than the work itself. We have seen systems that require three days and seven different teams to release a single small change that took three hours to create.

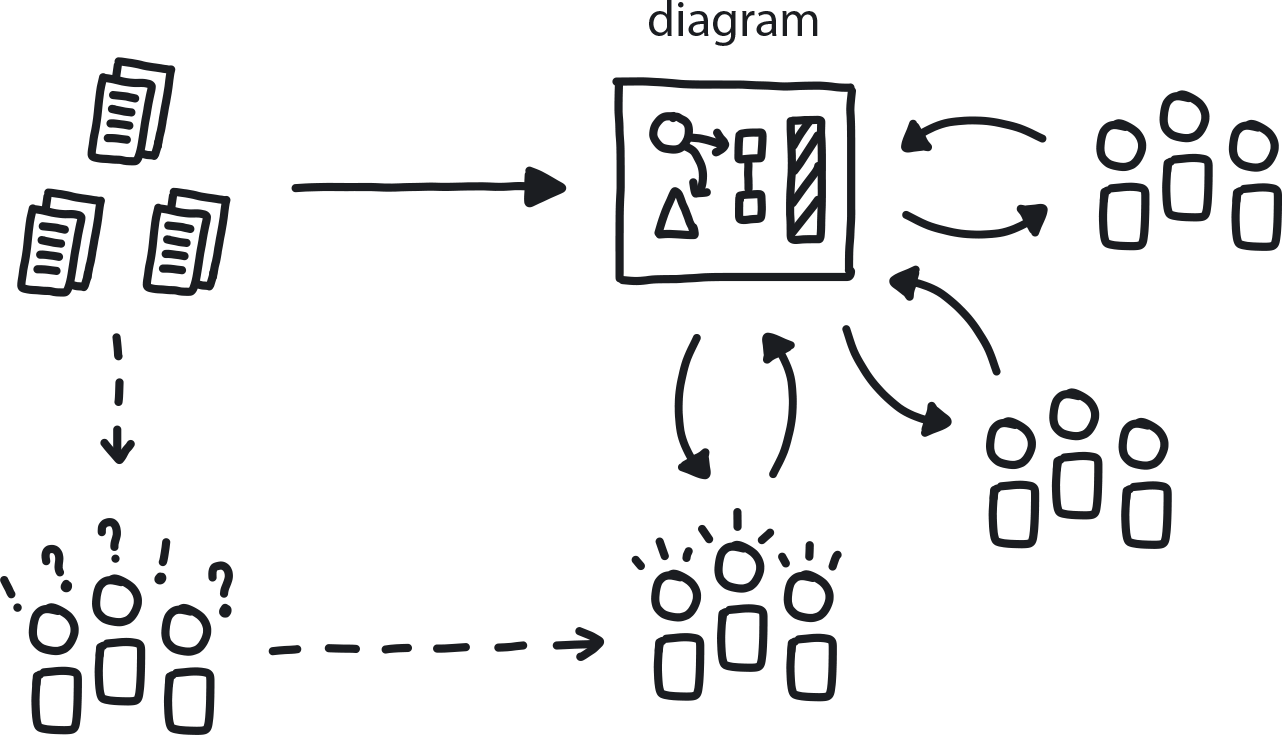
All of this eventually makes further system growth functionally impossible. The main, maybe only, fix would be to scale the whole system, even though only a small part of it is running out of resources—which requires buying very expensive hardware. (And then, over time, the same problems would simply re-emerge anyway.)

Distributed systems are much more complex to initially architect and implement, but once that initial work is invested they are much simpler to grow and evolve forward with improvements, new features, and changes. This is due to a much lower level of coordination overhead. That same three-hour code change task can simply proceed to production in 10 minutes without any human involvement at all.

A distributed system is any system whose components are executing independently, communicating and coordinating their actions by passing messages to one another. Historically these were on-prem servers, but systems began moving to the cloud. Cloud native then emerged as a software development approach that makes full and optimal use of the cloud computing model. Cloud native applications are built with modular service components—microservices—that communicate through APIs and get packaged in containers. The containers themselves are managed via an orchestrator (Kubernetes) and dynamically deployed on an elastic cloud infrastructure through Agile DevOps processes and continuous integration/continuous delivery (CI/CD) workflows.

OK, so that was a lot. Don’t worry, it’s only the general concepts we need to worry about right now—the high-level master blueprint for the system under construction. This leads us, aptly enough, to literally drawing one up.

**PATTERN: ARCHITECTURE DRAWING**



**Figure 11-15. Architecture Drawing**

A picture—or, in this case, a high-level outline sketch of your system’s basic architecture—can replace a thousand words, save time, and prevent misunderstandings.

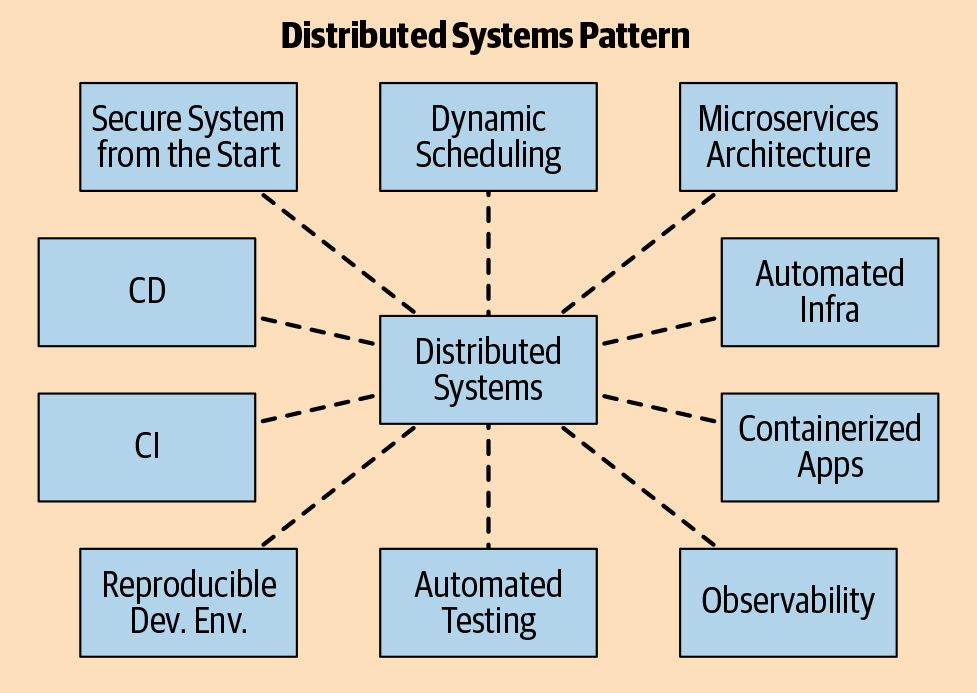
Distributed systems are inherently complex, which makes them difficult to visualize and describe, much less discuss. The Architecture Drawing is a classic No Regret Move, easy and inexpensive yet extremely beneficial. It’s also, thankfully, straightforward to do: produce a single, unified, and official version of the new cloud native system’s architecture for circulation and reference. This creates a simple and consistent visual that everyone on the Core Team should be able to draw, from memory, in less than 30 seconds.

A huge benefit to the Architecture Drawing is that it helps all stakeholders internalize a deep understanding of the architecture’s components and how they relate to each other. It also provides consistency of thought and makes discussion vastly easier.

With this clear blueprint of our distributed cloud native system in hand, let’s look at the patterns that comprise it. [Figure 11-16](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch11.html#the_distributed_systems_pattern) shows the technical, structural, and process-oriented patterns that go into designing cloud native distributed systems and underlie the Distributed Systems pattern:

* Microservices Architecture and Containerized Apps are fundamental pillars of cloud native architecture.
* Dynamic Scheduling, Automated Infrastructure, Automated Testing, and Observability describe these core technical aspects of a cloud native system.
* Continuous Integration (CI) and Continuous Delivery (CD) are the processes for implementing them.
* Secure From the Start Systems addresses how to keep everything safe.
* Reproducible Development Environment describes the workspace necessary for the Core Team to assemble all these pieces.

Thumbnail versions of all these patterns are available in [Appendix A](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/app01.html#library_of_patterns): Library of Patterns, which will direct readers to the full versions of each pattern in [Chapter 9](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch09.html#patterns_for_development_and_process), Patterns for Development & Process, and [Chapter 10](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch10.html#patterns_for_infrastructure_and_cloud), Patterns for Infrastructure & Cloud. Since we have discussed all of these technical aspects, particularly microservices, containers, and dynamic scheduling (a.k.a. Kubernetes), throughout the first part of this book, let’s leave the nitty-gritty details to the Core Team and keep the transformation design moving along at the level of architecture rather than detailed execution.



**Figure 11-16. The Distributed Systems pattern unpacks into essential elements of the cloud native architecture and development process.**

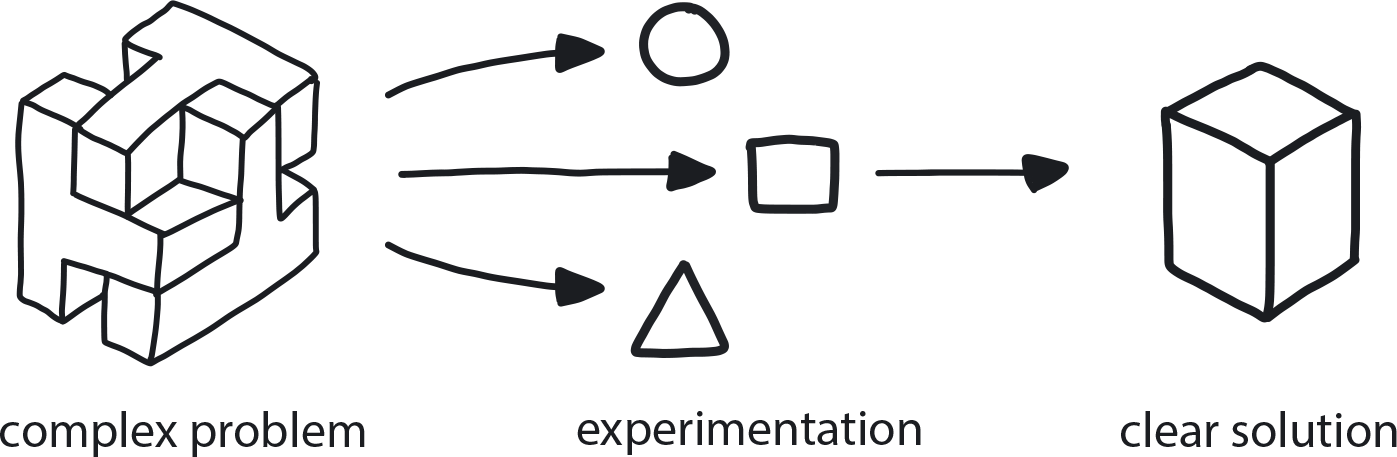
**Exploratory Experiments**

This is the practical first step in an organization’s technical transformation. At this point, the Core Team doesn’t yet have enough cloud native knowledge to identify the most general direction to head in—you can’t research unknown unknowns. So they begin by doing theoretical research: reading what others have done, case studies, white papers. Certainly doing No Regret Moves like trainings and other learning opportunities. There is no clear solution available because this is a complex problem and you cannot simply Google it—there are many solutions out there that sound great but simply don’t fit your particular use case.

The scope of the transformation is very wide. You need to collect many different puzzle pieces to create a consistent platform, and right now you don’t even know enough to understand what the right pieces even look like.

How do you find the right pieces? By doing experiments. Small, quick, low-cost investigations into different tools, techniques, and architectural combinations just to see what works and, more to the point, what does not. Core Team members learn enormously as they do this, building their understanding and skills. After a relatively short time, usually a month or two, this new knowledge and the experimentation process helps the team narrow down the field of options and uncover likely best-fit directions for meeting the transformation objectives.

**PATTERN: EXPLORATORY EXPERIMENTS**

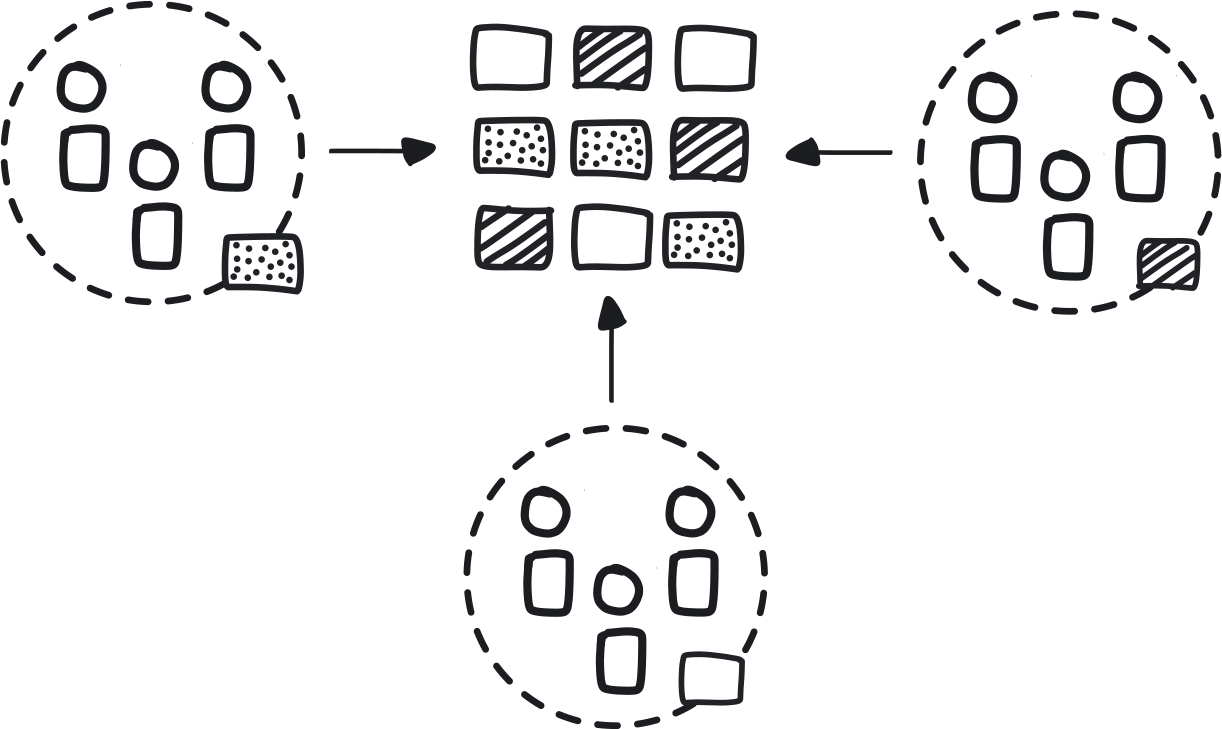


**Figure 11-17. Exploratory Experiments**

When dealing with a complex problem with no obvious available solution, run a series of small experiments to evaluate the possible alternatives and learn by doing.

This is the time of working with the fundamental building blocks of cloud native: Microservices Architecture, Containerized Apps, Continuous Integration, Continuous Delivery, and different types of automation. We will start with Microservices Architecture, because that drives everything else.

**PATTERN: MICROSERVICES ARCHITECTURE**



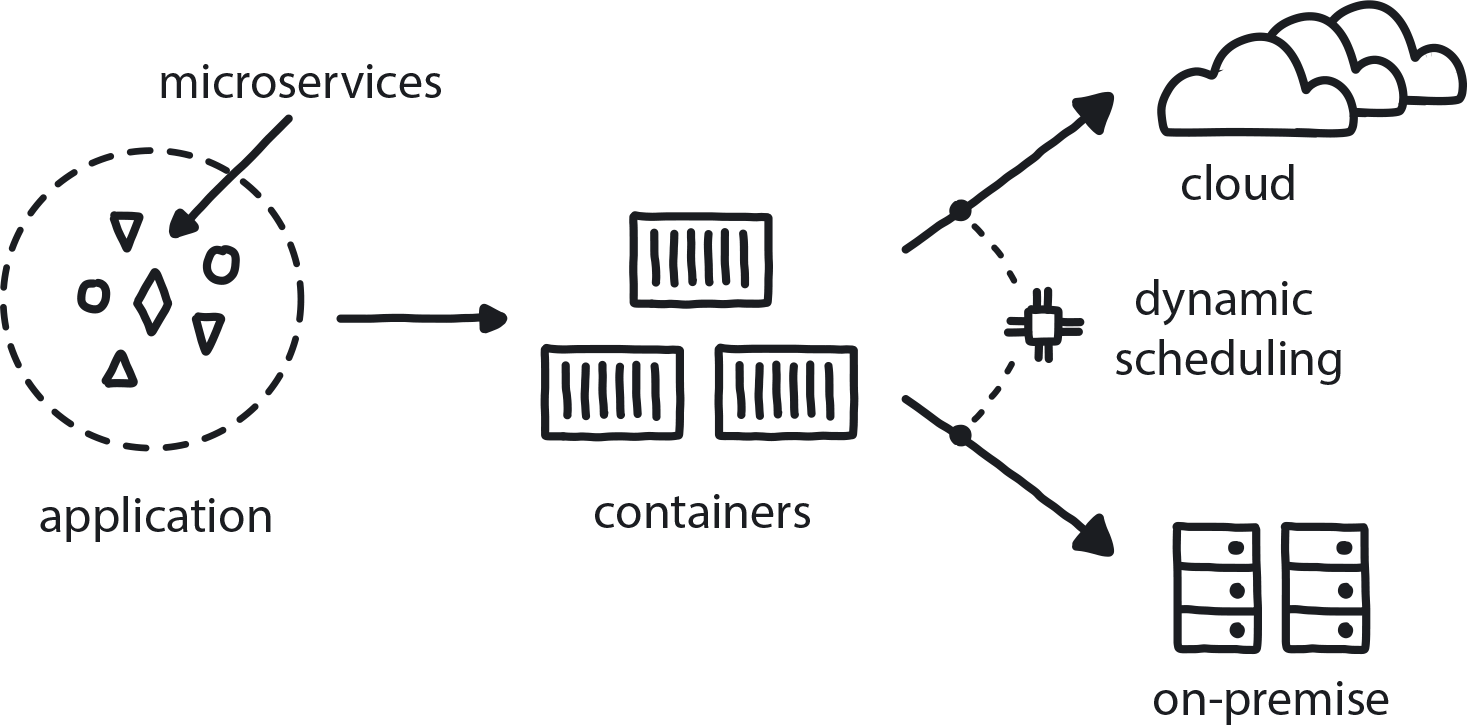
**Figure 11-18. Microservices Architecture**

To reduce the costs of coordination between teams delivering large monolithic applications, design software as a suite of modular services that are built, deployed, and operated independently.

Microservices are an implementation of distributed-systems architecture where an application is delivered as a set of small, modular components that are completely independent from each other and communicate via APIs. This is an extremely flexible, resilient, and scalable way to build software, but also a very complicated one. In the cloud native way of working, a team is responsible for delivering one service, from design to delivery, and then supporting it as it runs even as they iterate forward with small improvements.

To run the application, all of its microservices get packaged into a container. Containers are lightweight, standalone executable software packages that include everything required to run an application: code, runtime, system tools, libraries, and settings. They are a sort of “standard unit” of software that packages up the code with all its dependencies so it can run anywhere, in any computing environment.

**PATTERN: CONTAINERIZED APPS**

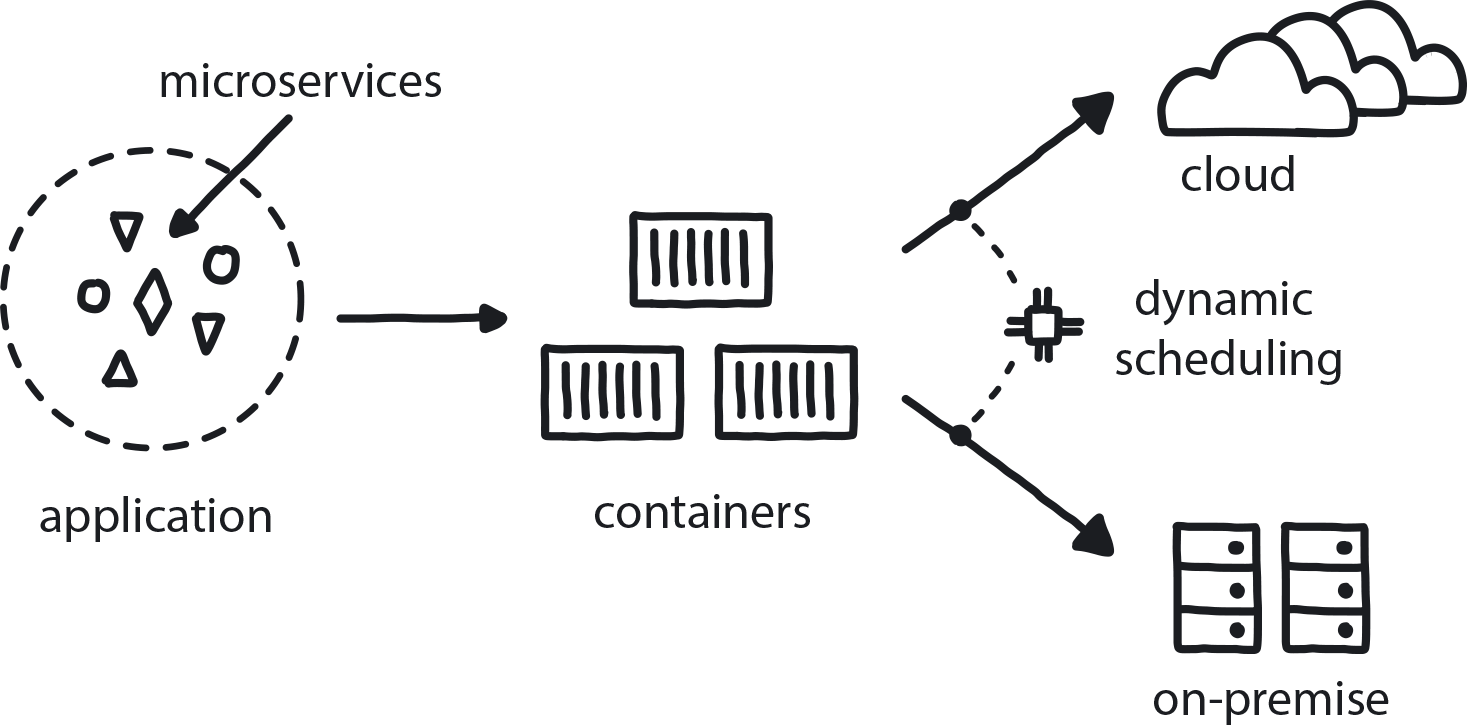


**Figure 11-19. Containerized Apps**

When an application is packaged with all its necessary dependencies, it does not rely on the runtime environment and so can run agnostically on any platform.

To run in the cloud, whether public or private, the containerized microservices need to be organized, managed, and told when and where to run. This is the job of the Dynamic Scheduler, or orchestrator (Kubernetes is by far the most well known and widely used but is far from the only orchestration option out there).

**PATTERN: DYNAMIC SCHEDULING**

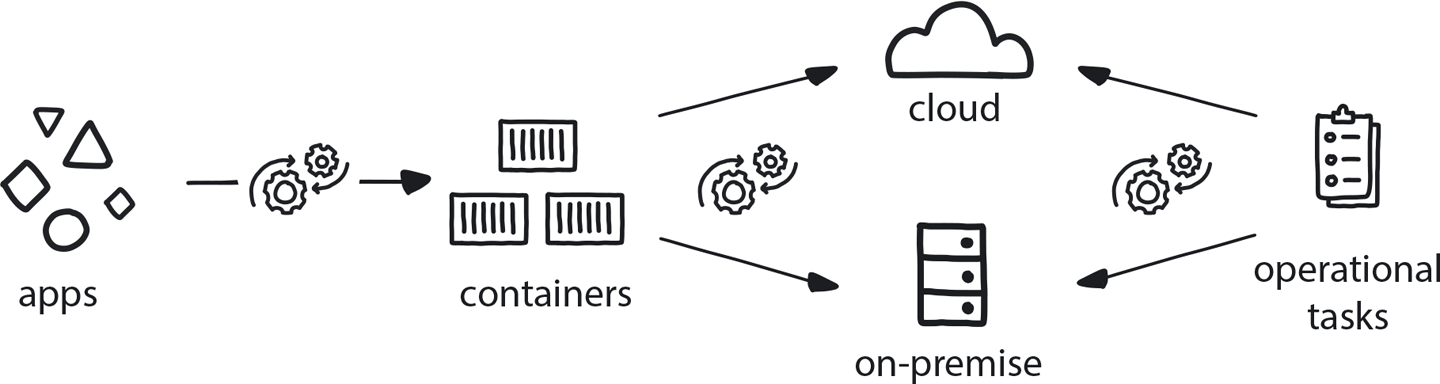


**Figure 11-20. Dynamic Scheduling**

An orchestrator (typically Kubernetes) is needed to organize the deployment and management of microservices in a distributed, container-based application to assign them across random machines at the instant of execution.

Building software that is optimized for the cloud is complicated, but it does come with a few helpful advantages. Since there are so many moving pieces now, far more than humans can manage or keep up with, cloud native computing requires full automation. Automated Infrastructure handles server provisioning, configuration management, builds, and deployments and monitoring at top speed and without the need for manual intervention. Similarly, Automated Testing ensures that all the testing required to take any product change to production is built into the development process.

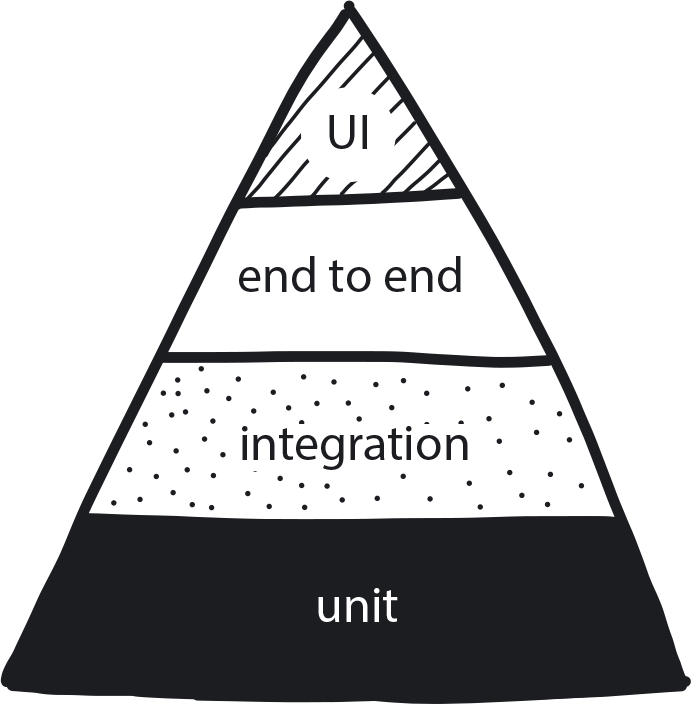
**PATTERN: AUTOMATED INFRASTRUCTURE**



**Figure 11-21. Automated Infrastructure**

The absolute majority of operational tasks need to be automated. Automation reduces inter-team dependencies, which allows faster experimentation and leads in turn to higher development velocity.aut

**PATTERN: AUTOMATED TESTING**



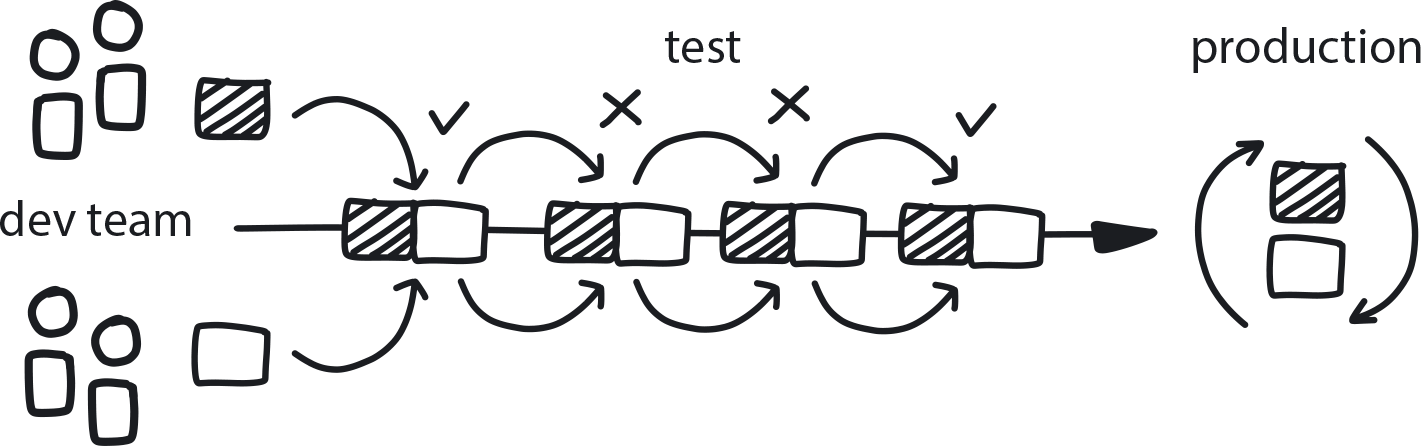
**Figure 11-22. Automated Testing**

Shift responsibility for testing from humans (manual) to an automated testing framework so developers can deliver faster and have more time to focus on improving features to meet customer needs.

To make the most of automation, the Core Team must adopt Continuous Integration and Continuous Delivery as their workflow.

Continuous Integration (CI) is a coding philosophy and set of practices that drive development teams to implement small changes and check in code to version-control repositories frequently. The technical goal of CI is to establish a consistent and automated way to build, package, and test applications that are always highest quality, and always ready to be released.

**PATTERN: CONTINUOUS INTEGRATION**

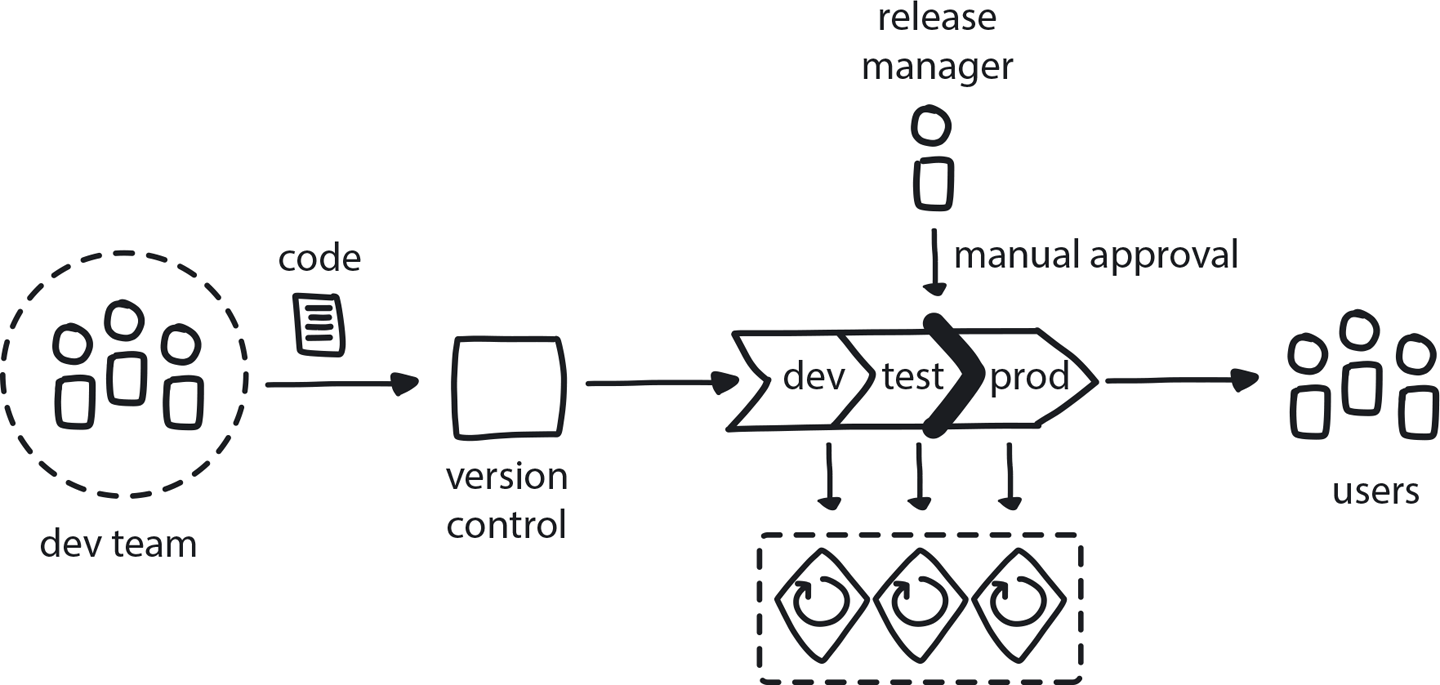


**Figure 11-23. Contiguous Integration**

Frequent integration of small iterative changes speeds overall delivery and increases the quality of the code.

Continuous Delivery(CD) starts where Continuous Integration ends. CD automates the delivery of applications to selected infrastructure environments. Most teams work with multiple environments outside the production pipeline, such as development and testing environments, and CD ensures there is an automated way to push code changes to them. CD automation then performs any necessary service calls to web servers and databases, and executes procedures when applications are deployed.

**PATTERN: CONTINUOUS DELIVERY**



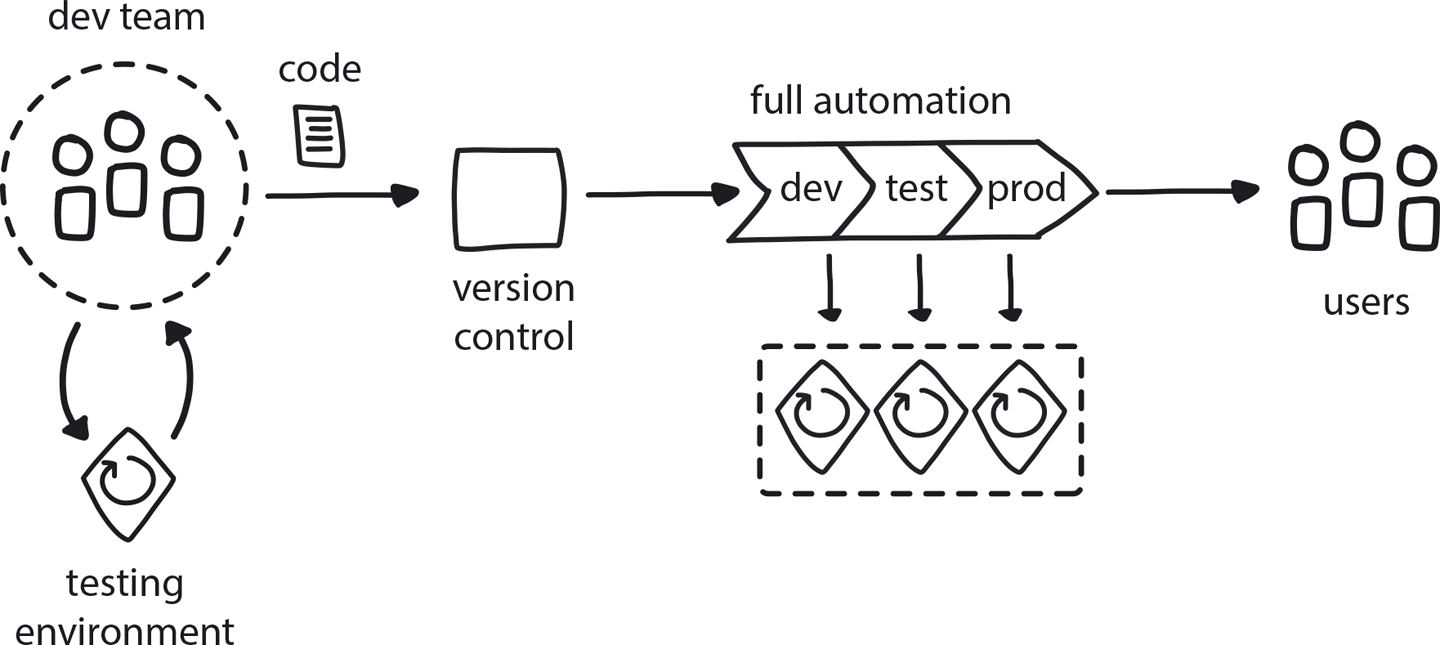
**Figure 11-24. Continuous Delivery**

Keeping a short build/test cycle means code is always ready for production and features can be immediately released to customers—and their feedback quickly returned to developers.

Better together: Continuous Integration and Continuous Delivery are so integral both to each other and to the cloud native approach that they are usually used as a single term. CI/CD used together make it possible to implement continuous deployment, where application changes run through the CI/CD pipeline. Builds that pass Automated Testing get deployed directly to production environments with Automated Infrastructure. This happens frequently, on a daily—or even hourly—schedule.

CI and CD are difficult to achieve, however, unless developers are able to test each change thoroughly. The Reproducible Dev Environment pattern defines the conditions for where the Core Team can practice all of these brand new tools and techniques while applying CI/CD processes.

**PATTERN: REPRODUCIBLE DEV ENVIRONMENTS**



**Figure 11-25. Reproducible Dev Environments**

Developers need to test their daily work in an environment that is easy to spin up and that matches production tooling as closely as possible.

The Core Team has now learned how to orchestrate the containerized microservices they’ve been experimenting with, using automation and cloud native methodologies. What comes next?

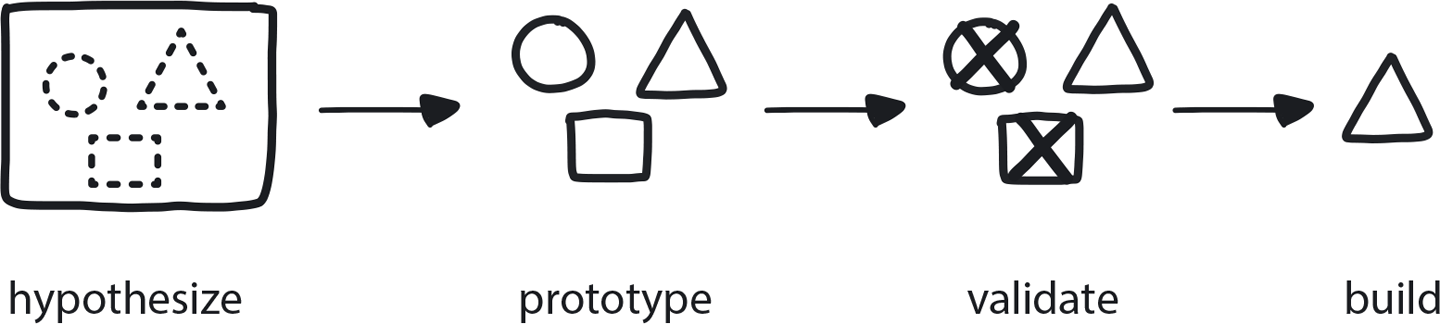
**Proof of Concept**

So, great! Your research has given you a better understanding of what is going on, but there is still fairly high risk involved in making decisions at this point. How do you know when it’s time to stop Exploratory Experiments and take the next step?

The trick is to look at it as a balance situation. You are weighing the need for more certainty against the need to move things forward. The moment of truth comes when a few likely looking paths begin to stand out from the cloud native wilderness of choices. You still need to gather more information—a lot more—but you have achieved the milestone of identifying some promising areas where you need your information to be deeper.

At this point it’s time to move to the next stage of the process: deepening understanding of the most promising areas. It’s time for the Proof of Concept (PoC) pattern.

**PATTERN: PROOF OF CONCEPT (POC)**



**Figure 11-26. Proof of Concept**

Before fully committing to a solution that can significantly affect the future, build a small prototype to demonstrate viability and gain understanding.

You still don’t know enough to make any kind of large commitment at this point, whether to a vendor or an open-source solution. Any decision right now carries massive risk because full commitment to one path means you will continue to build further functionality on top of this solution. The further you go, the harder it becomes to change course.

You have, however, narrowed the field to a few potential transformation paths that appear promising. It is time to investigate these different options for architecture and toolsets through deeper, though still not prohibitively difficult, Proof of Concept exercises. The stakes are a little higher now: PoCs are more complicated than the simple experiments the Core Team has been running thus far, so costs and time investment do go up. Thus, if your best-guess PoC fails there is some pain, but it’s not a terrible tragedy. You can revert if needed (not to mention gain valuable insight into what doesn’t work).

PoCs are fun! A real chance for the engineers to roll up their sleeves and play with these cool cloud native toys by choosing one of the potential solutions the Core Team has identified through experiments and building a basic prototype to demonstrate viability—or lack thereof. Remember that it’s still OK to fail.

**POCS: ELIMINATING WEAK OPTIONS**

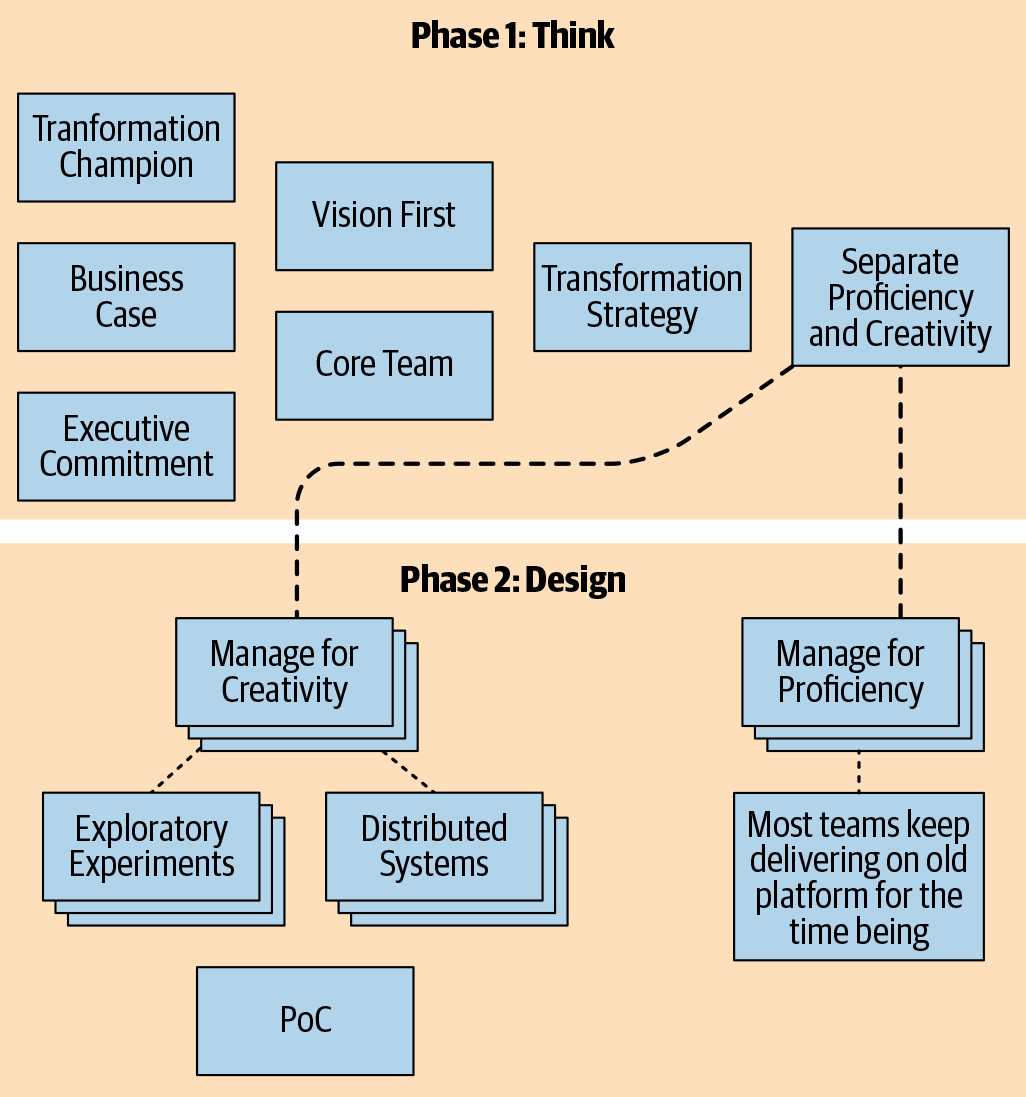
Something to keep in mind during the PoCs phase is how young the cloud native ecosystem still is. Even now there is still a lot of uncertainty and knowledge to be gained, and there are also still significant gaps in the toolset. Cloud native right now is similar to the birth of the automotive industry at the beginning of the 20th century, when there were more than 100 automobile manufacturers in the United States alone. Now there are fewer than 20 major manufacturers worldwide, and who has heard of most of the ones that dropped away?

So, there will be shakeout and consolidation. Every month major new tools get introduced into the cloud native ecosystem, and they don’t always fit each other well. The entire point of PoCs is to choose 10 to 20 of those tools and try to see how they work together. Among the safest bets to choose from are in the Cloud Native Computing Foundation’s [*“Graduated Projects”*](https://www.cncf.io/projects/). These are well-established tools like Kubernetes, which are likely to be around for the long run, and there are many related/supporting tools and resources built around them.

Quick and dirty is the theme with PoCs, which should take a few days to a few weeks at most to build the most primitive version you can get away with. Focus only on hard issues related to the specific needs of the current project, and stop when the solution is clear—don’t worry about creating a nice UX, just make the darn thing work. If possible, try to actually demonstrate something functional so you can collect information about how this solution will affect the business.

But, most of all, be ready to throw it away when you are finished proving your point. This doesn’t mean that you must absolutely trash all your hard work. But you probably should. Very often when initial quality is intentionally low, it’s cheaper and easier to just scrap it and rebuild. This way you get a clean slate and the chance to do things right, applying all those hard-earned PoC lessons you just learned. Otherwise, you will probably have to spend a lot of time fixing all the stuff you did wrong before you can even start building anything new on top.

[Figure 11-27](https://learning.oreilly.com/library/view/cloud-native-transformation/9781492048893/ch11.html#phase_2__design_cloud_nativ) shows the map of patterns we’ve covered in Phase 2: Design, from the point where the Core Team began researching the best possible design for the company’s cloud native transformation. The team members are being managed for creativity, with freedom to experiment and fail if necessary, while the rest of the company keeps on delivering on the existing system. The Core Team does Exploratory Experiments to investigate all the aspects of Distributed Systems that will go into the new cloud native platform, until their learning curve has flattened and they’ve identified some good candidates for more in-depth investigation through PoCs.



**Figure 11-27. Phase 2: Design cloud native transformation design patterns**

Because once you’ve picked a clear winner—once the PoC process has mapped the likely right transformation path—it’s time to get ready to build for real!

*So far in our design for a cloud native transformation, we have covered getting ready for the transition (including figuring out if you really even need to do one in the first place!). Next we had a detailed look at how to de-risk the transformation by experimenting your way to the right path, using a gradual and iterative approach. Now that PoCs have revealed the likely best platform, we will move on to Part 2 for applying patterns to do the actual implementation.*